Spécification UML du contrôle d'accès dans les systèmes d'information : Une approche coopérative de la conception des rôles dans un modèle RBAC

THÈSE

présentée et soutenue publiquement le 23 Mai 2003
en vue de l'obtention du

Doctorat de l'Université d'Artois
et Doctorat de l'Ecole Polytechnique Silesienne
(Spécialité Informatique)

par

Aneta Poniszewska

Composition du jury

Président :

Monique Picavet, HDR, Maître de Conférences à l'Université de Lille1

Rapporteurs :

Danielle Boulanger, Professeur à l'Université de Lyon3
Tadeusz Czachorski, Professeur à l'Ecole Polytechnique Silesienne

Examinateurs :

Gilles Goncalves, Professeur à l'Université d'Artois (Directeur de thèse)
Piotr Szczepaniak, Professeur à l'Ecole Polytechnique de Lodz (codirecteur de thèse)
Ewa Napieralska-Juszczak, Maître de Conférences à l'Université d'Artois
Fred Hemery, Maître de Conférences à l'Université d'Artois

Université d'Artois - Laboratoire de Génie Informatique et d'Automatique d'Artois (ex LABOGP)
rue de l'Université, 62408 Béthune Cedex
Tél : +33 (0)3 21 63 71 64 - Fax : +33 (0)3 21 61 17 80

Remerciements

Je tiens à remercier mon directeur de thèse, Gilles Goncalves qui m'enseigna l'ABC d'une méthode de recherche, pour m'avoir accueilli dans le laboratoire, pour ses conseils et sa patience.

Fred Hemery pour l'encadrement qu'il a assuré tout au long de cette thèse, ses remarques pertinentes, sa patience et la disponibilité dont il a toujours fait preuve.

Piotr Szczepaniak pour qui m'enseigna la rigueur d'un travail, pour ses conseils et pour son aide.

Madame Danielle Boulanger, professeur à l'Université de Lyon3 et Monsieur Tadeusz Czachorski, professeur à l'Ecole Polytechnique Silesienne, pour avoir accepté d'être rapporteurs de cette thèse et pour les remarques et conseils éclairés qui ont contribué à améliorer ce manuscrit.

Madame Monique Picavet, habilité à diriger des recherches à l'Université de Lille1 pour avoir étudié ce travail et accepté de participer à ce jury.

Je remercie Ewa Napieralska-Juszczak pour Tout.

Les enseignants du laboratoire pour leur accueil depuis mon premier séjour à l'université.

Ma famille et les amis pour leur patience,

plus tous ceux que j'ai oubliés qui je l'espère me pardonneront.

Table des matières

Table des figures

Liste des tableaux

Liste des algorithmes

Introduction

Avec la mondialisation des échanges, l'entreprise gagnante se repose de plus en plus sur la réactivité de son système d'information. L'accroissement et la complexité des fonctionnalités auxquelles il doit répondre actuellement font que sa conception et sa réalisation sont des activités professionnelles difficiles et stratégiques pour l'entreprise. Pour cela, il est nécessaire de disposer de modèles, de méthodes, de techniques et d'outils qui font du développement de systèmes d'information une véritable activité d'ingénierie au même titre que le génie mécanique, le génie civil, etc.

Dans les années 90 sont apparues les méthodes d'analyse et de conception orientées objet. Dans cette catégorie on peut citer le langage unifié - UML (Unified Modeling Language) - langage crée par J. Rumbaugh, G. Booch et I. Jacobson qui résulte de la fusion de trois méthodes proposées antérieurement par ces auteurs (OMT - Object Modeling Technique, le méthode de Booch, OOSE - Object Oriented Software Engineering) [BRJ98, Gro00a, Res00].

Le système d'information automatisé étant une pièce importante de l'organisation et du bon fonctionnement de l'entreprise, il importe de développer un système de sécurité qui le protège contre toutes menaces internes ou externes.

La sécurité de l'information étant un sujet large à aborder nous nous intéressons essentiellement au contrôle d'accès aux données, c'est-à-dire à la sécurité logique. Les dispositions de sécurité logique protègent les ressources immatérielles (données, programmes, etc.) contre les erreurs et les malveillances. Un système sûr est un système qui, à travers l'utilisation de dispositifs spécifiques de sécurité, permet de contrôler l'accès à l'information de telle manière que seuls les individus habilités, puissent lire, écrire, créer ou détruire de l'information.

Toutes les personnes habilitées (utilisateurs) n'ont pas accès à toutes les fonctionnalités d'une application. Un utilisateur habilité à utiliser une fonctionnalité de l'application (par exemple : consulter des comptes) n'est pas automatiquement autorisé à consulter toutes les données (par exemple : les comptes de salaires). Ces objectifs étant définis, il convient de déterminer un niveau de sécurité (contrôle d'accès) optimal qui soit un juste compromis entre des exigences de sécurité.

Nous nous intéressons plus particulièrement à la notion de rôle au niveau du contrôle d'accès aux données et services du système d'information. Le contrôle d'accès basé sur la notion de rôle représente une alternative intéressante par rapport aux modèles traditionnels comme : le modèle discrétionnaire (DAC - Discretionary Access Control) [CFMS94, San94, SJ93] trop complexe à gérer ou le modèle mandataire (MAC - Mandatory Access Control) [BRJ98, San94, SJ93], trop rigide à mettre en place.

Le modèle du contrôle d'accès basé sur la notion de rôle - le modèle RBAC - a été défini dans les années 90 [SCFY96]. Le modèle RBAC définit l'accès aux données d'un utilisateur en se basant sur les actions que cet utilisateur peut exécuter dans le système. Il demande l'identification d'un ensemble de rôles dans le système. Un rôle est un ensemble d'actions et de responsabilités nécessaires à une activité dans un système.

Définir les rôles d'un modèle RBAC, revient à déterminer les rôles de chacun au sein d'une organisation. Des modèles existent mais quand est il au niveau de leur mise en place au sein d'un système d'information ?

Peu de recommandations existent concernant le contrôle d'accès au niveau des méthodes de conception des systèmes d'information comme la méthode Merise [Gab98, KMPRS98] ou la méthode Ossad [CS97]. De même, aucune méthodologie n'est donnée au niveau des méthodes dédiées à la sécurité des systèmes d'information comme la méthode Marion [Lam91] ou la méthode Méhari [Clu], pour spécifier et mettre en place un modèle basé sur les rôles.

Seuls quelques travaux de recherche ont abordé l'ingénierie des rôles [TOB98, FH97, ES99, JSpSB98], mais ceux-ci sont bien souvent déconnectés de la conception et de l'évolution du système d'information.

L'objectif de ce travail de thèse est d'intégrer l'aspect sécurité au niveau de la conception (et de son évolution) d'un système d'information. On se base sur une modélisation orientée objet en utilisant comme support le langage de modélisation UML.

Nous focaliserons essentiellement notre travail sur le contrôle d'accès aux données et aux services du système d'information.

Nous voulons proposer dans cette thèse un outil qui permettra à l'administrateur de sécurité de gérer plus simplement un contrôle d'accès basé sur la notion de rôle dans un système d'information.

Nous proposons donc un processus de création des rôles d'un système d'information basé en deux étapes. Pendant la phase de conception du système d'information, le concepteur va générer de façon quasi automatique les rôles adaptés aux fonctionnalités du système décrites dans les diagrammes UML. Dans un second temps, c'est-à-dire durant la phase de mise en exploitation du système d'information, l'administrateur de sécurité va affecter les rôles précédents aux utilisateurs en se basant sur les responsabilités de ceux-ci et sur la politique globale de sécurité définie par l'entreprise.

Pour ce faire, nous avons proposé une extension du modèle RBAC de R. S. Sandhu permettant plus de flexibilité dans l'organisation des rôles. Nous avons validé notre approche sur une plateforme UML-XML-Java que nous avons développée et qui utilise l'AGL de Rational Rose pour la conception UML d'une application à intégrer dans le système d'information de l'entreprise.

Le processus d'administration de la sécurité d'un système d'information est une tâche très complexe. Des contraintes de sécurité doivent être définies pour respecter une politique de sécurité donnée. Dans notre approche, des contraintes peuvent être définies par le concepteur au niveau d'une nouvelle application, et par l'administrateur de sécurité (au niveau du système global) pour intégrer l'application dans le système existant. Pour garantir la cohérence globale de l'ensemble, c'est-à-dire du nouveau système d'information, nous avons donc proposer un mécanisme de validation de ces contraintes basé sur le modèle RBAC étendu. Un outil d'administration intégrant ce mécanisme est actuellement en cours d'implémentation.

La Figure 1 présente le plan de ce mémoire.

Le *Chapitre 1* est consacré à la définition d'un système d'information et de son système de sécurité. Nous regardons les méthodes et langages utilisés pour la mise en place d'un système d'information, et ensuite les méthodes dédiées pour la sécurité des systèmes d'information.

Le *Chapitre 2* présente la notion de contrôle d'accès dans un système d'information. Il décrit les différentes politiques de contrôle d'accès et leurs modèles. On se focalise, en particulier, sur un modèle basé sur la notion de rôle - le modèle RBAC. Nous examinons aussi les travaux de recherche portant sur les méthodologies centrées sur l'ingénierie des rôles.

Dans le *Chapitre 3*, nous proposons une extension au modèle RBAC et nous décrivons le rapprochement des concepts du modèle RBAC avec certains concepts UML. On présente un processus de création des rôles dans un système d'information basé sur une séparation des responsabilités en deux niveaux.

Dans le *Chapitre 4*, nous présentons la vérification de la cohérence globale du modèle RBAC à l'aide de contraintes. On donne une classification des contraintes. On décrit la confrontation des deux niveaux de responsabilités et la chaîne de validation qui garantit la cohérence globale du système d'information au niveau sécurité.

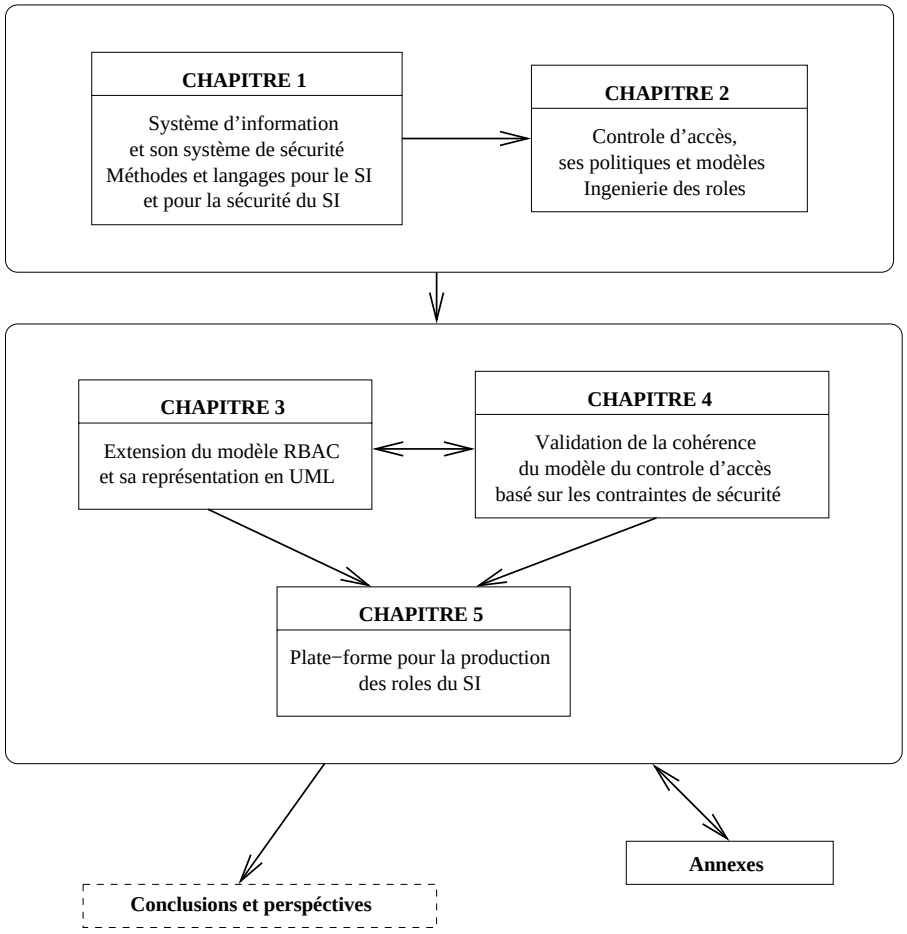


FIG. 1 – Le plan du mémoire.

Le *Chapitre 5* présente une plate-forme pour la production des rôles d'un système d'information ainsi que l'outil d'administration en cours de développement.

Des *Conclusions et perspectives* terminent le manuscrit de cette thèse.

On trouvera finalement en annexes :

- une description de notre exemple de l'application, i.e. l'application "Gestion des Notes" (*Annexe A*),
- un extrait du méta-modèle d'UML qui contient seulement les diagrammes de paquetages utilisés par la suite de notre travail (*Annexe B*),
- une description détaillée de la méthode de traduction des contraintes décrite dans le Chapitre 4 (*Annexe C*),
- un extrait de la syntaxe du langage XML (*Annexe D*),
- une description d'une structure de document résultat de la plate-forme UML-XML-Java (*Annexe E*).

Chapitre 1

La sécurité des systèmes d'information

Objectifs du chapitre :
- *Rappeler les principes d'organisation d'une entreprise*
- *Définir le système d'information*
- *Introduire les méthodes et langages utilisés pour la mise en place d'un système d'information*
- *Décrire la sécurité des systèmes d'information*
- *Regarder les méthodes dédiées pour la sécurité des systèmes d'information*

L'entreprise d'aujourd'hui se doit d'être, plus réactive et plus flexible pour s'adapter aux multiples évolutions qui perturbent les marches mondiaux. Son organisation et son système d'information associé devront faire l'objet d'une ou plusieurs phases de réorganisation pour passer avec succès ces évolutions. La définition des rôles au sein d'une organisation va être fortement affectée par ces fréquents changements.

Aujourd'hui les systèmes d'information couvrent de nombreuses applications financières, industrielles, militaires et commerciales qui font majoritairement appel à l'informatique. Ces applications manipulent une très grande quantité d'informations stockées numériquement.

Ces différents moyens de traitement de l'information doivent être combinés pour mieux répondre aux besoins de leurs futurs utilisateurs et se protéger efficacement contre des actions malveillantes, ou des sinistres.

Un utilisateur mal intentionné peut en effet accéder, diffuser, modifier et voire même détruire des informations confidentielles et stratégiques pour une organisation. Les conséquences de ces attaques sont nombreuses. Elles vont du plan individuel (information strictement personnelle) au plan national (information liée à la défense) en passant par le plan industriel (information financière ou stratégique).

L'objectif de la sécurité informatique est de réduire les risques associés à l'utilisation des systèmes d'information, à un niveau acceptable pour une organisation.

Ce chapitre dresse un bilan sur les recommandations, les méthodologies existantes associés à un système d'information.

Après un rappel sur l'organisation des entreprises, nous avons regardé dans un premier temps, le rôle du système d'information dans l'entreprise et les méthodes, les langages utilisés pour la conception ou le développement des systèmes d'information. Nous avons choisi de présenter la méthode Merise [Mél72, TNH75, TRC86, Gab98, KMPRS98] et la méthode Ossad [CS97].

Dans un second temps nous allons présenter un survol général de la sécurité des systèmes d'information. Elle définit les objectifs de la sécurité, les critères généraux de la sécurité et leurs

domaines d'application et présente une structure organisationnelle pour la réalisation et la gestion de la sécurité des systèmes d'information. On regarde enfin les méthodes dédiées la sécurité des systèmes d'information - la méthode Marion [Lam91] et la méthode Méhari [Clu].

1.1 Les structures des entreprises

L'objectif de l'entreprise est de répondre à un besoin impératif de survie. Pour cela elle doit produire et vendre des biens ou des services, pour un client qui en a exprimé le besoin tout en dégageant des profits lui permettant de prospérer. L'entreprise dispose de ressources humaines, technologiques, financières, qu'elle doit organiser et utiliser au mieux pour atteindre ses objectifs et maximiser ses profits.

Une bonne organisation est basée sur des relations adéquates entre les personnes qui travaillent dans une entreprise et les fonctions à remplir par celle-ci. Il faut définir le rôle de chacun en fonction de ses compétences et des objectifs de l'entreprise. Choisir une bonne organisation est assez complexe, parce que il y a plusieurs types d'organisations possibles. On peut la représenter sous plusieurs points de vue pour simplifier sa description.

1.1.1 La structure fonctionnelle d'une entreprise

Une ***fonction*** est un regroupement d'activités de même nature. La fonction devient dans les petites entreprises synonyme de "service".

On distingue principalement quatre grandes fonctions de l'entreprise (Figure 1.1) :
- *conception* - *concevoir* (ou développer) un bien ou service en réponse à une demande,
- *achat* - *acheter* les matériaux indispensables à la fabrication,
- *production* - *produire* ce bien ou service et
- *vente* - le *vendre* sur le marché.

Ces fonctions génèrent différentes tâches qui sont accomplies par un ou plusieurs acteurs de l'entreprise. Elles impliquent également la mise en action d'autres fonctions, plus spécifiques. Toutes ces fonctions s'articulent entre elles, se coordonnent en fonction de la politique générale de l'entreprise et de son environnement.

Les éléments caractéristiques de la structure fonctionnelle d'une entreprise sont présentés dans la Figure 1.2.

1.1.2 La structure organique d'une entreprise

Par analogie avec un organisme vivant, une entreprise peut être vue comme un système complexe composé d'***organes*** qui interagissent et remplissent des fonctions vitales à sa survie.

Elle est donc caractérisée par :
- une **structure** qui décrit la manière dont les parties d'un tout sont arrangées entre elles,
- des **organes** (ou unités, ou postes) qui sont des assemblages de moyens humains, matériels et incorporels de l'entreprise,
- des **fonctions** qui sont nécessaires à la réalisation du projet de l'entreprise.

Les différences entre organe et fonction sont les suivantes :

1. *organe* est une réalité physique qui permet de répondre aux questions classiques : quoi, qui, où, combien,
2. *fonction* est une réalité abstraite qui permet de répondre aux questions : comment, quand, pourquoi.

On peut distinguer trois types d'unités (ou postes ou services) :

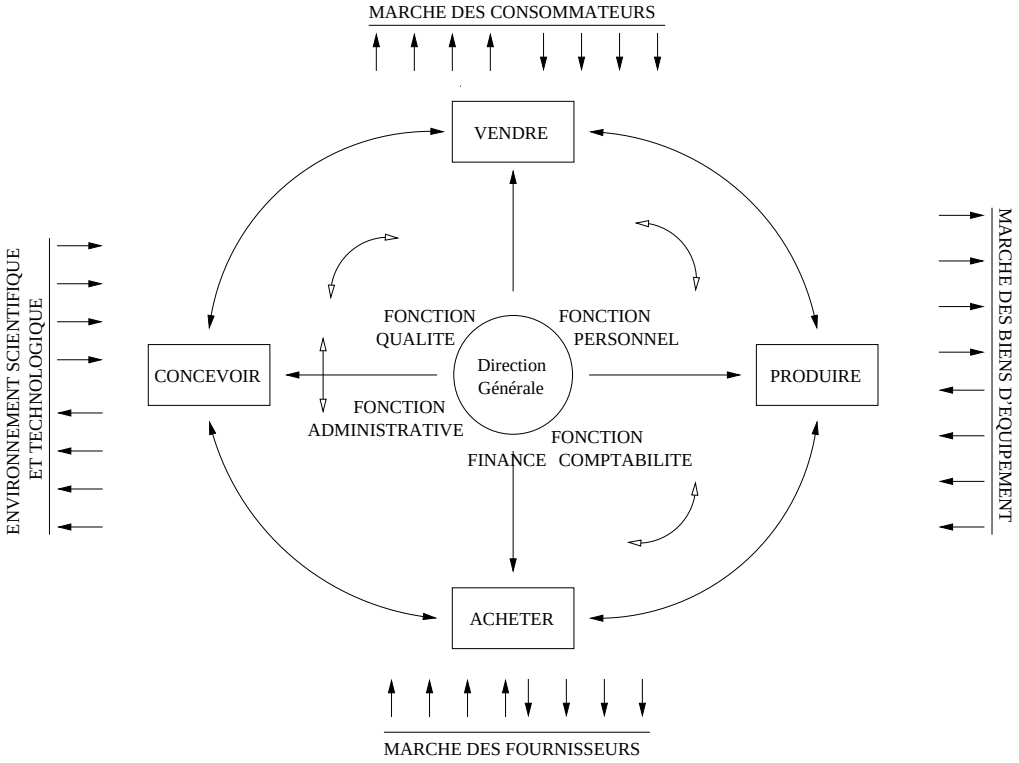


FIG. 1.1 – Une structure dynamique de fonctionnement de l'entreprise [FLLP91].

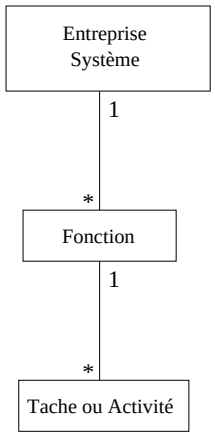


FIG. 1.2 – La vue fonctionnelle d'une entreprise.

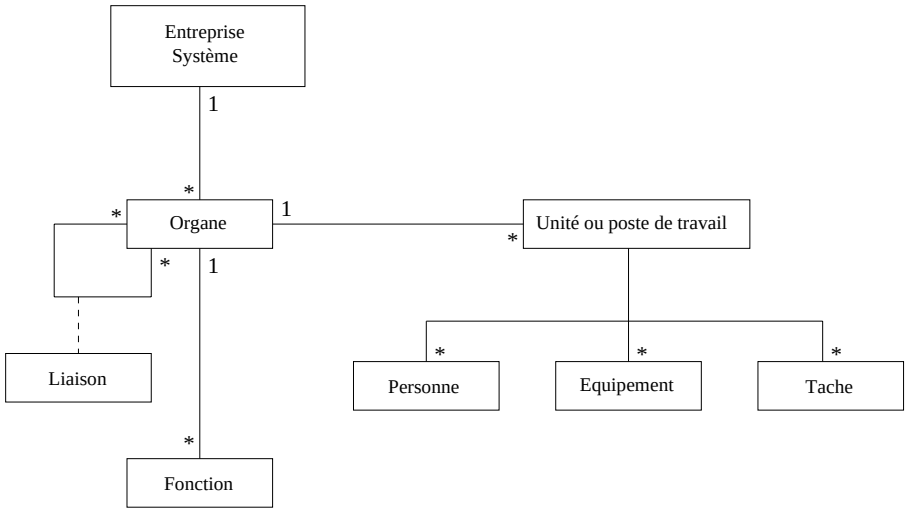


FIG. 1.3 – La vue organique d'une entreprise.

- ***unités opérationnelles*** qui réalisent des tâches essentielles de l'entreprise comme : la Conception, la Production, la Distribution, etc.
- ***unités fonctionnelles*** qui assistent dans leurs tâches quotidiennes les unités précédentes (Service, Conseil, Gestion, etc.),
- ***unités managerialles*** qui coordonnent l'ensemble des unites.

Pour pouvoir se coordonner, chaque unité ou poste a besoin d'entretenir les liaisons avec la reste de l'entreprise. Les ***liaisons*** expriment les relations professionnelles entre les postes. Elles définissent les canaux de communication formelle au sein de l'organisation.

Les éléments caractéristiques de la structure organique d'une entreprise sont présentés dans la Figure 1.3.

1.2 Le système d'information

Le système d'information d'une entreprise apparaît de plus en plus comme un élément stratégique qui doit permettre d'améliorer sa compétivité face à la mondialisation des marchés. Avec le développement de la technologie d'Internet, nombreuses sont les directions informatiques à ouvrir leurs systèmes d'information vers leurs partenaires privilégiés (fournisseurs, sous-traitants, clients) par la mise en place de projets Extranet, de places de marché. Il découle de cette ouverture que la sécurité de ces systèmes d'information est de plus en plus mise à l'épreuve.

1.2.1 Les fonctionnalités d'un système d'information

Un ***système d'information*** est un ensemble organisé de ressources : matériels, logiciels, personnels, données, procédures d'acquisition, de traitement, de stockage, de communication des informations (sous forme de données, de textes, images, sons, etc.) et de relations qui les coordonnent par rapport aux missions de l'entreprise (Figure 1.4).

1.2.2 Les types de systèmes d'information

On peut distinguer différents types de systèmes d'information selon leur finalité [Rei99] :

1. *systèmes supports d'opérations* - au niveau opérationnel - ils ont comme finalité d'assister le traitement des opérations quotidiennes correspondant aux activités courantes de l'entreprise ; ils sont centrés sur le court terme,

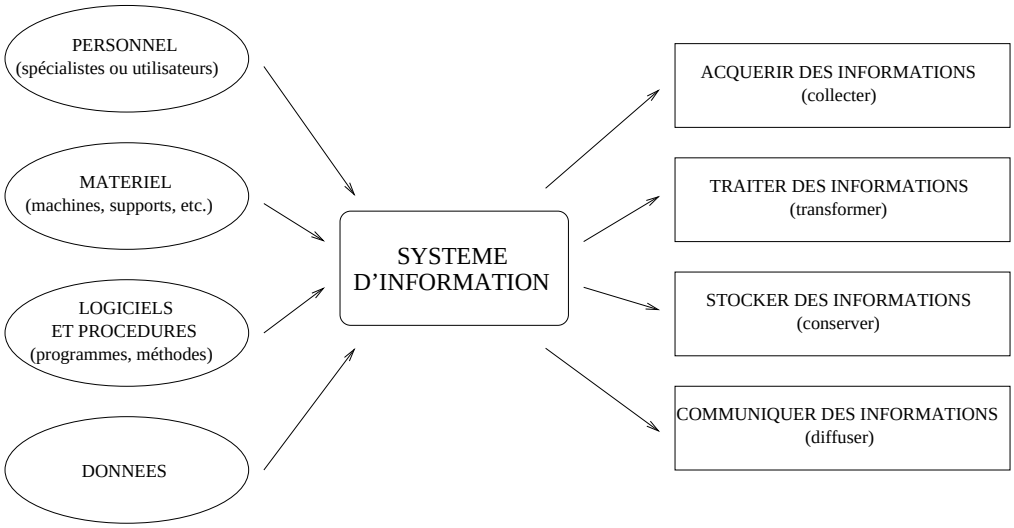


FIG. 1.4 – Les composants du système d'information.

2. *systèmes supports de gestion* - au niveau tactique - ils ont pour objectif principal d'aider les cadres et dirigeants de différents niveaux à prendre des décisions opportunes ; on y trouve : des systèmes de gestion de production (GPAO - Gestion de Production Assistée par Ordinateur, ERP - Entreprise Ressources Planning), des systèmes de planification (APS - Advanced Planning System) ; ils sont centrés sur le moyen terme,
3. *systèmes supports de stratégie* - au niveau stratégique - on y trouve des outils d'analyse, de prévisions et d'aide à la décision (ECR - Efficient Custom Response, datamining) permettant d'assurer le développement de l'entreprise ; ils sont surtout centrés sur le long terme.

1.2.3 La conception d'un système d'information

La structure d'un système d'information est généralement complexe et doit s'inscrire dans une logique rigoureuse de gestion de projet.

La Figure 1.5 présente les étapes de la réalisation d'un système d'information, depuis le cahier des charges du client qui définit le problème jusqu'à la maintenance.

1.3 Les méthodes de conception des systèmes d'information

Concevoir le système d'information issu d'une organisation est une tâche complexe qui nécessite une compréhension précise du fonctionnement de celle-ci pour le modéliser efficacement.

Deux approches de modélisation sont souvent mises en opposition :

1. **approche analytique** (cartésienne) - procède par découpage de l'organisme (i.e. système) en sous-systèmes et étudie le comportement de chaque sous-système indépendamment des autres,
2. **approche systémique** - le système est vu comme un tout et les interactions qui existent entre sous-systèmes sont cette fois ci prises en compte.

La modélisation systémique issue de la théorie générale des systèmes en biologie [Moi77] permet de voir l'entreprise comme :

1. un ***système vivant*** composé d'organes (ses ressources humaines, matérielles et financières) destinés à remplir des fonctions vitales et organisé suivant une certaine logique qui lui permet d'attendre un objectif connu, sa survie,

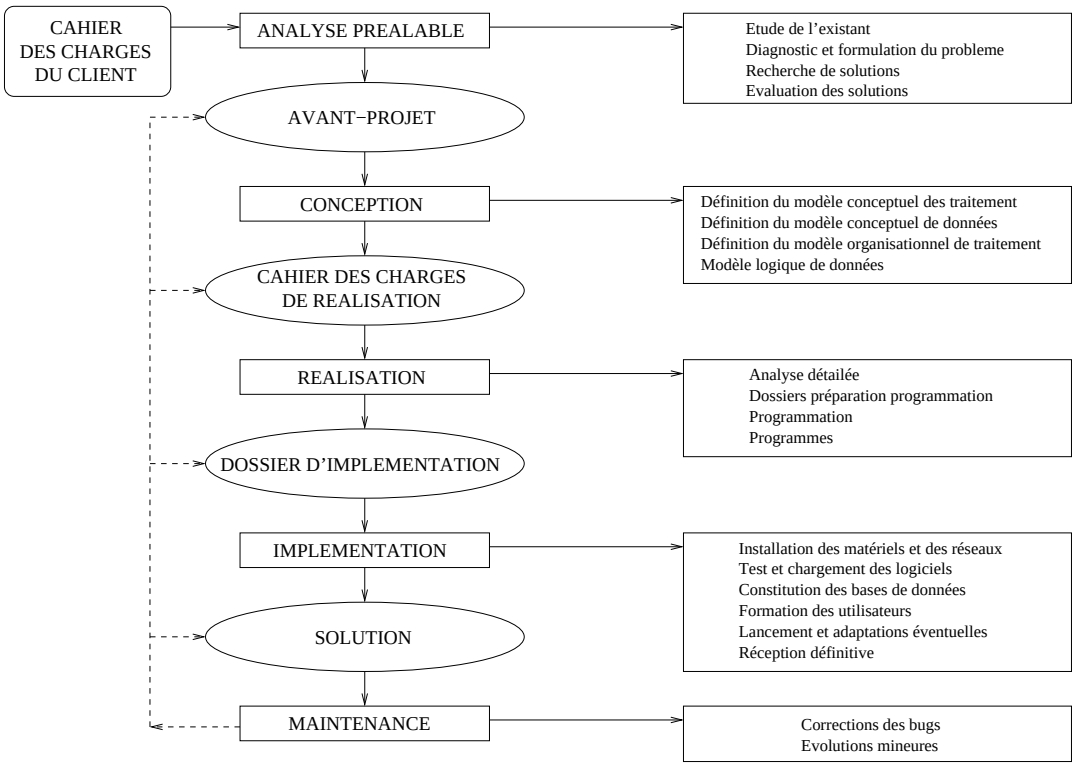


FIG. 1.5 – Les étapes de la réalisation d'un système d'information.

2. un ***système ouvert***, qui possède une frontière plus ou moins perméable lui permettant d'interagir avec son environnement (ses partenaires, ses clients).

L'intérêt majeur de l'approche systémique réside dans le fait qu'elle est à la fois dynamique et globale. Dans la pratique, une combinaison des deux approches est utilisée lorsque le système à modéliser est à la fois complexe et compliqué [Moi77].

Nous présentons dans cette section deux méthodes de conception des systèmes d'information - les méthodes Merise et Ossad qui s'appuient sur cette approche.

1.3.1 La présentation de la méthode Merise

Merise [Mél72, TNH75, TRC86, Gab98, KMPRS98] est situé dans le domaine des méthodes de conception et de développement des systèmes d'information. Cette méthode s'est imposée comme un standard depuis les années 80. Elle est enseignée dans toutes les formations en systèmes d'information.

D'une part Merise présente des modèles conceptuels du système d'information et d'autre part elle propose une démarche méthodologique de conception et de développement de ces modèles.

1.3.1.1 Les niveaux de description d'un système d'information

Merise réponse principalement sur un processus de description qui sépare les données et les traitements.

Merise propose de décrire un système d'information suivant différents niveaux d'abstraction. A chaque niveau correspond une préoccupation différente du système d'information sur la description

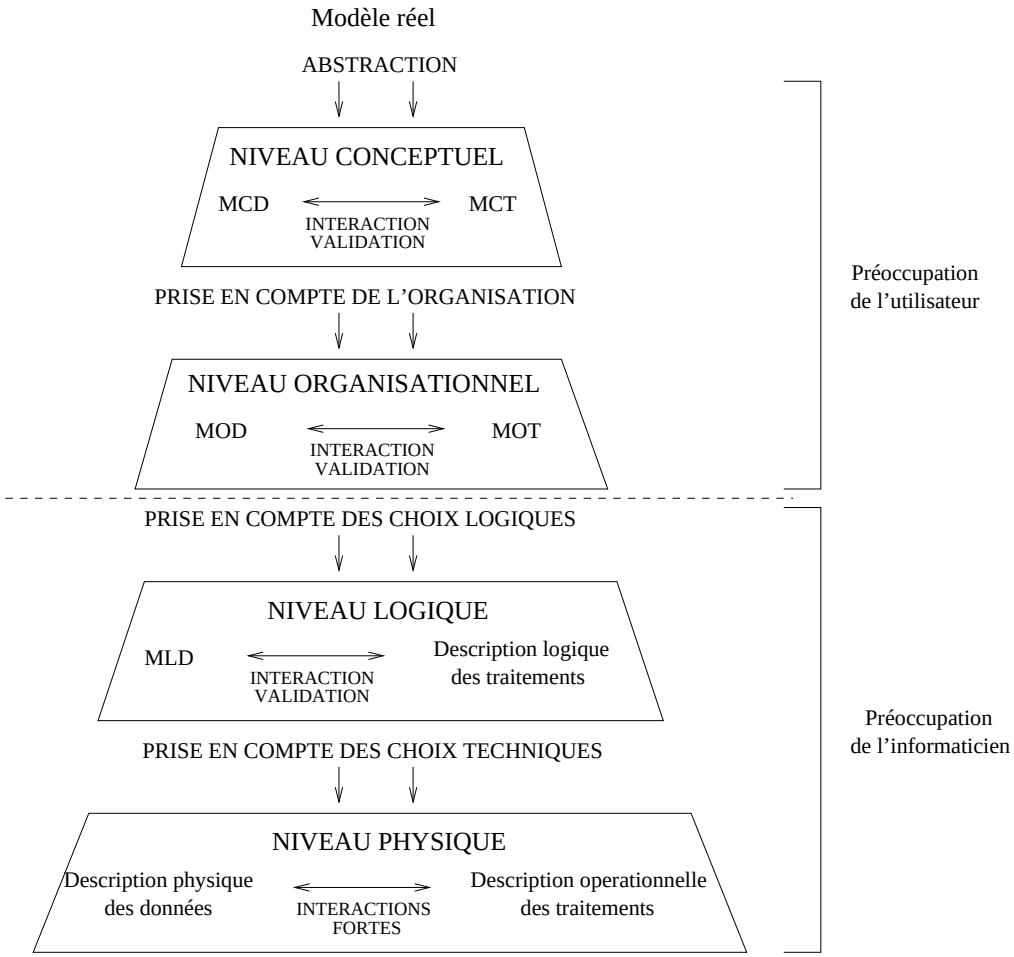


FIG. 1.6 – Les niveaux de description dans la conception d'un système d'information [Gab98].

des données et des traitements. Ces niveaux de description peuvent être groupés en deux ensembles [Gab98] (Figure 1.6) :

- le *niveau conceptuel* et le *niveau organisationnel* sont indépendants des aspects techniques liés à l'informatisation,
- le *niveau logique* et le *niveau physique* prennent en compte la technologie informatique de la solution retenue.

Le **niveau conceptuel** correspond aux finalités de l'entreprise. Il s'agit de décrire le "quoi" en faisant abstraction des contraintes techniques et d'organisation. Les modèles utilisés pour la description conceptuelle du système d'information sont : le *Modèle Conceptuel de Données (MCD)* et le *Modèle Conceptuel des Traitements (MCT)*.

Le **niveau organisationnel** décrit "qui fait quoi et où". Il définit les traitements entre l'homme et la machine, le mode de fonctionnement (temps réel ou temps différé) et l'affectation des données et des traitements par type de site organisationnel et par type de poste de travail. Les modèles définis dans ce niveau sont : le *Modèle Organisationnel de Données (MOD)* et le *Modèle Organisationnel des Traitements (MOT)*.

Le **niveau logique** permet de décrire la conception technique en terme d'unités de traitements (temps réel ou temps différé) et prend en compte la structuration propre au stockage informatisé

Niveau de description	Concepts manipulés	
	Données	Traitements
CONCEPTUEL	objets (entités) relation propriétés MCD	processus opération événement (résultat) synchronisation MCT
ORGANISATIONNEL	objets (entités) relation propriétés MOD	procédure phase tache MOT
LOGIQUE	table attribut MLD	procedure phase tache fonction (module) DLT
PHYSIQUE	fichier rubrique DPD	application unité de traitement (temps réel) DOT

FIG. 1.7 – Les concepts de Merise [Gab98].

des données. On distingue : le *Modèle Logique de Données (MLD)* et la *Description Logique des Traitements (DLT)*.

Le **niveau physique** définit le choix des outils techniques. Il spécifie : la *Description Physique des Données (DPD)* et la *Description Opérationnelle des Traitements (DOT)*.

Ces deux niveaux, logique et physique, décrivent le "comment".

La Figure 1.7 présente les concepts manipulés dans les quatre niveaux décrits ci-dessus.

1.3.1.2 Le processus de développement d'un système d'information

Associée à ces modèles, Merise propose une méthodologie de développement qui couvre tout le cycle de vie du système d'information [TRC86, Gab98] :

- *étude préalable* - propose une architecture globale de la solution en tenant compte des orientations de gestion, d'organisation et de choix techniques validés par le comité directeur du projet ; cette étape produit le dossier d'étude préalable,
- *étude détaillée* - décrit la solution à réaliser, c'est-à-dire les données saisies, modifiées et restituées, et les traitements exécutés sur ces données ; elle contient deux phases : la conception générale et la conception détaillée ; elle produit le dossier de spécifications détaillées,
- *réalisation* - contient l'étude technique, la production de logiciel et les tests qui permettront de valider le logiciel obtenu,
- *mise en oeuvre* - exécute toutes les actions (formation, documentation, installation des matériels, initialisation des données, etc.) qui permettront d'aboutir au lancement du système auprès des utilisateurs,
- *maintenance* - pour les évolutions du système d'information.

Objet de la modéliastion	Type de modèle	Concept principal
objectifs	modèle abstrait	fonction
moyens	modèle descriptif	role

TAB. 1.1 – Les niveaux de modélisation ossadiens.

La construction d'un système d'information d'une entreprise consiste à bien identifier ses domaines d'activité (i.e. ses processus) et les différents flux qui les traversent. Celle-ci est généralement précédée d'une phase de réorganisation qui permet à l'entreprise de remettre à plat ses processus donc son organisation.

Un ***flux*** est la représentation de l'échange d'informations entre deux activités ou entre une activité et un partenaire extérieur (i.e. acteur) à l'entreprise.

Un ***acteur*** est une entité organisationnelle identifiable par les missions qu'il doit remplir dans le cadre du champ d'étude défini. Un acteur dont les missions se situent à l'intérieur du champ d'étude est un *acteur interne*. Un acteur dont les missions se situent en dehors du champ d'étude mais qui échange des informations avec un acteur interne, est un *acteur externe*.

1.3.2 La présentation de la méthode Ossad

La méthode **Ossad** (***Office Support Systems Analysis and Design*** - *Analyse et conception de systèmes d'assistance pour le bureau*) [CS97] résulte d'un projet de recherche et de développement dans le cadre du programme ESPRIT (European Strategic Program for Research in Information Technology). C'est une méthode qui permet d'abord d'analyser l'état d'une organisation, puis de la faire évoluer (de procéder à une réorganisation) en collaboration avec toutes les personnes concernées par celle-ci. Cette méthode a été créée en complément aux méthodes de développement d'application informatiques classiques comme Merise.

L'approche Ossad distingue deux niveaux d'abstraction ou de modélisation (Tableau 1.1) :
- **abstrait** - pour exprimer les ***objectifs*** d'une organisation,
- **descriptif** - pour exprimer ses ***moyens*** (humains et technologiques).

A ces deux niveaux d'abstraction correspondent deux concepts principaux : la ***fonction*** et le ***rôle***. Les autres concepts sont présentés dans la Figure 1.8.

Le modèle abstrait

Le modèle abstrait s'intéresse aux missions, aux buts ou aux objectifs de l'organisation, indépendamment des moyens - humains et techniques - mis en oeuvre pour les réaliser. Il se préoccupe du "quoi" et du "pourquoi" et non du "comment", "qui" et "quand". Les modèles abstraits sont relativement invariants par rapport à l'organisation mise en place.

Le modèle descriptif

Un modèle descriptif décrit les moyens mis en oeuvre par l'organisation, pour atteindre les objectifs exprimés dans le modèle abstrait. Il répond au "comment", "qui" et "quand". Il s'intéresse à la répartition des tâches, à l'ordonnancement des opérations, au matériel et aux logiciels utilisés.

Pour décrire les éléments de modélisation plusieurs supports graphiques existent (Figure 1.9).

Au niveau du modèle abstrait, on trouve le graphe de modèle abstrait dont les concepts sont [CS97] :
- ***fonction*** - un sous-ensemble de l'organisme étudié poursuivant des objectifs homogènes. Une fonction peut être décomposée en un certaine nombre de sous-fonctions, dans le but de mieux la définir et ces sous-fonctions peuvent être décomposées, etc. jusqu'à une activité,
- ***fonction externe*** - une entité externe à l'organisme étudié, faisant partie de son environnement et poursuivant des objectifs homogènes,

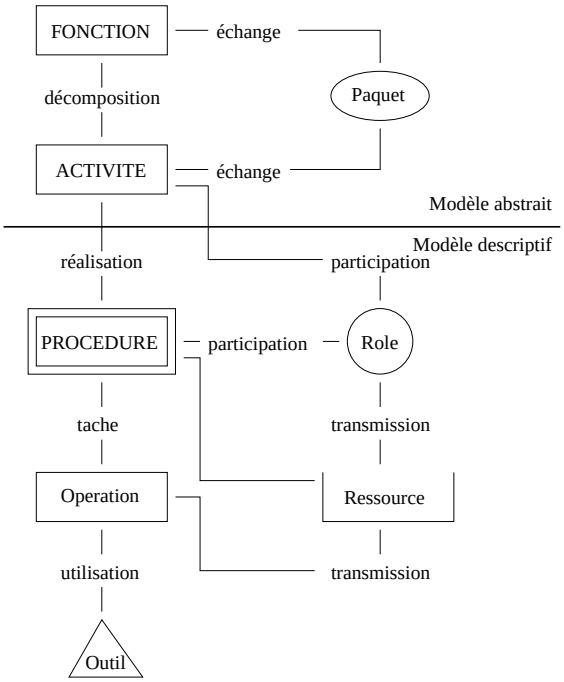


FIG. 1.8 – Les concepts de deux modèles [CS97].

- ***paquet*** - un ensemble d'informations circulant entre les fonctions et/ou les fonctions externes,
- ***activité*** - un sous-ensemble de l'organisme étudié représentant le niveau le plus fin de la hiérarchie de décomposition d'une fonction.

Au niveau du modèle descriptif, nous trouvons les graphes qui sont présentés dans la Figure 1.10 [CS97] :

Une **matrice activités-rôles** - un tableau, dont chaque ligne correspond à une activité du modèle abstrait et chaque colonne un rôle du modèle descriptif ; elle constitue la réponse à la question : "qui fait quoi" ; elle formalise la liaison entre les deux niveaux de modélisation, l'abstrait et le descriptif - au moins un acteur jouant le rôle spécifié intervient dans l'accomplissement de l'activité concernée.

Un ***rôle*** - un ensemble de responsabilités confiées à un ou plusieurs acteurs dans le déroulement d'une procédure. La définition du rôle met l'accent sur la distinction entre celui-ci et l'acteur - un ***acteur*** est une personne physique, jouant un ou plusieurs rôles et un rôle peut être joué par un ou plusieurs acteurs. Un ***rôle externe*** - un ensemble de responsabilités appartenant à un ou plusieurs acteurs extérieurs à l'organisation, dans le déroulement d'une procédure. Une ***équipe*** - un ensemble homogène de rôles (i.e. agissant vers un but commun) qui permettra à un ensemble d'acteurs de jouer en son sein un rôle complémentaire.

Le modèle descriptif s'intéresse aussi à la circulation des informations entre les rôles et les procédures ; il est nécessaire d'introduire deux nouveaux concepts : une ***ressource*** qui représente des informations sur un support physique et une ***procédure*** qui décrit la manière de réaliser une activité.

Un **graphe de rôles** - permet de décrire la circulation des ressources entre les rôles, les rôles externes et les équipes ; une ressource peut être émise (produite) par un et un seul rôle (interne, externe ou équipe) et être utilisée (consultée ou modifiée) par un et un seul rôle (interne, externe ou équipe).

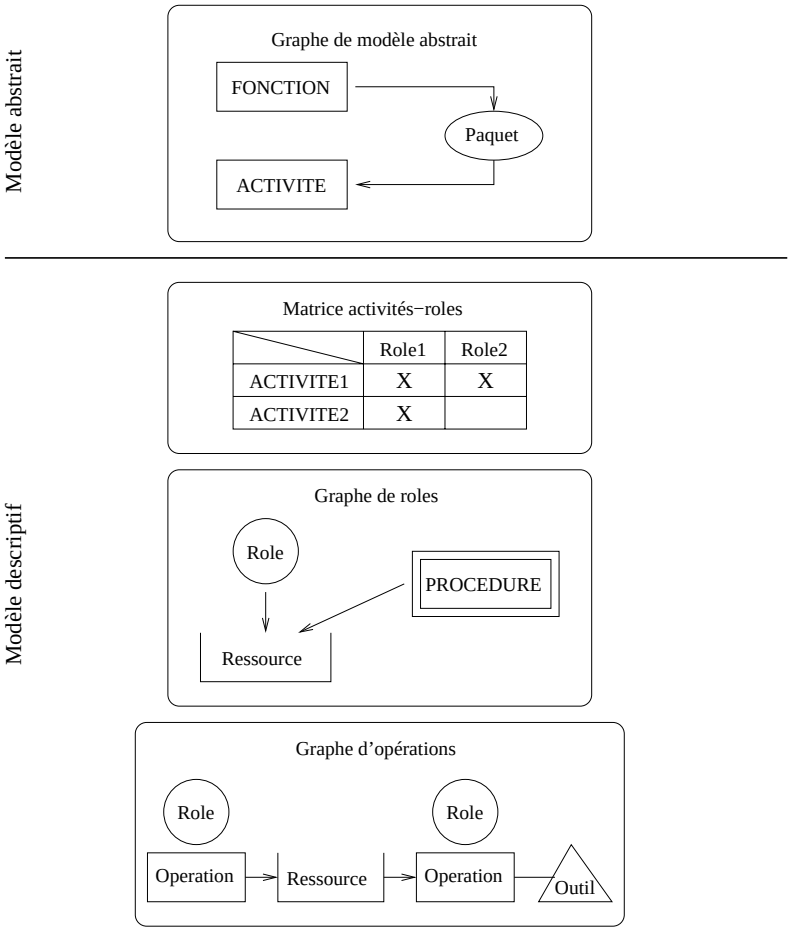


FIG. 1.9 – Le formalisme graphique d'Ossad.

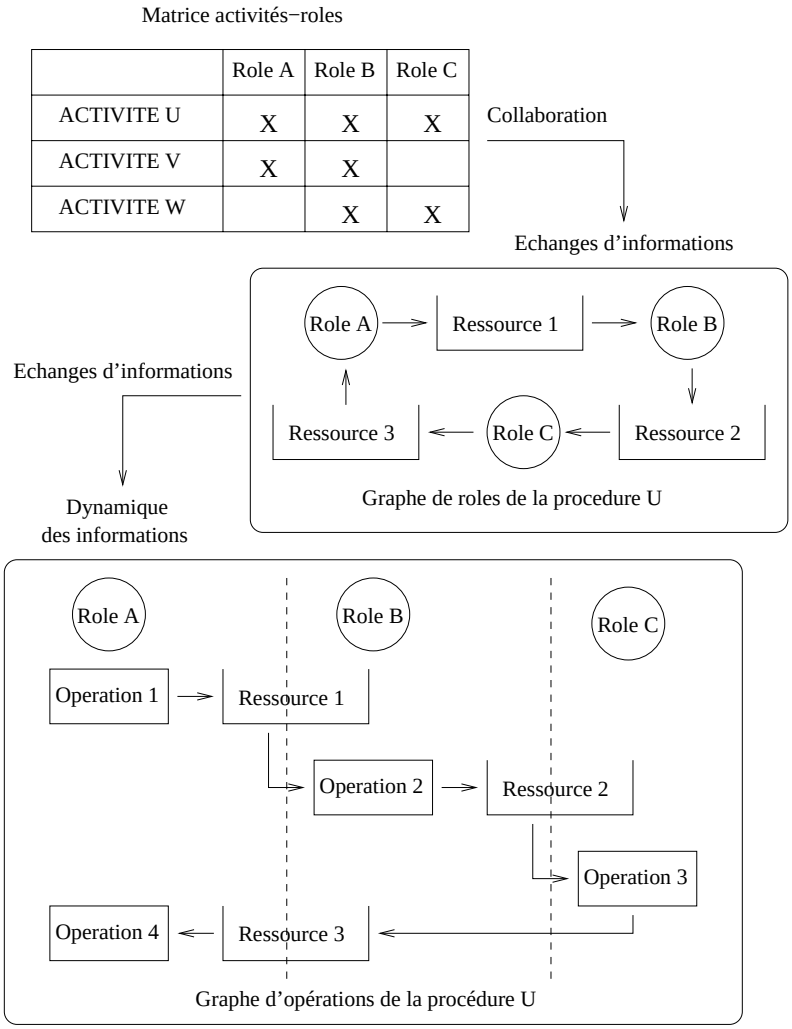


FIG. 1.10 – Les liens entre matrice activités-rôles, graphe de rôles et graphe d'opérations [CS97].

Par rapport à une matrice activités-rôles, un graphe de rôles constitue une formalisation plus détaillée des rôles de l'organisation. Alors que la matrice établit uniquement la liste des rôles collaborant au sein des procédures (activités du modèle abstrait), le graphe précise cette contribution en indiquant les informations effectivement échangées entre les rôles pour l'accomplissement des procédures.

Un **graphe d'opérations** permet de représenter la circulation de ressources entre les opérations d'une même procédure. Un tel graphe décrit donc : des opérations, des ressources et des rôles. Une ressource peut être émise par une et une seule opération et une opération est utilisée par un et un seul rôle.

Les concepts suivants correspondent à des objets propres à l'organisation.

Une ***unité*** est un découpage administratif de l'organisation, comme par exemple le "département marketing", la "section des contentieux", le "unité comptable".

Un ***poste***, par exemple "chef du personnel", "adjoint administratif", est défini par un contrat de travail entre une unité et un acteur, celle-ci mettant à disposition de celui-ci l'environnement nécessaire pour qu'il puisse jouer l'ensemble des rôles qui lui sont confiés. Un poste est donc un ensemble de rôles joués par le même acteur.

1.3.3 Comparaison des méthodes

Les deux méthodes sont toutes deux basées sur une approche systémique de l'entreprise. Merise est plus dédiée à la conception des systèmes d'information en offrant un modèle logique adapté aux bases de données relationnelles. Ossad est davantage tourné sur les aspects de l'organisation de l'entreprise avec des concepts proches (fonction, rôle, ressources) de ceux rencontrés en entreprise.

Le modèle conceptuel de Merise et le modèle abstrait d'Ossad constituent tous deux des outils bien adaptés pour représenter l'activité de l'entreprise indépendamment de son organisation. Ossad est un outil de modélisation réellement puissant et exploitable par tout un chacun, que ce soit de manière active (par le personnel chargé de formaliser les possibilités de réorganisation) ou de manière passive (par l'ensemble du personnel concerné par cette réorganisation et désirant participer à son élaboration, ainsi qu'à sa future mise en oeuvre).

1.4 La sécurité des systèmes d'information

Ces dernières années, le recours étendu aux technologies de l'information a fortement amélioré les performances des systèmes d'information, mais a aussi contribué du même coup à augmenter leur vulnérabilité.

La sécurité d'un système d'information peut être définie comme sa non-vulnérabilité à des accidents ou à des attaques volontaires, c'est-à-dire l'impossibilité que ces agressions produisent des conséquences graves sur l'état et le fonctionnement du système.

Pour une entreprise la *sécurité de son système d'information* consiste à limiter les pertes financières qui résulterait d'une défaillance (i.e. sinistre) d'un ou plusieurs composants de ce système.

1.4.1 Objectif de la sécurité

L'objectif de la ***sécurité des systèmes d'information*** est de garantir qu'aucun préjudice ne puisse mettre en péril la pérennité de l'entreprise. Cela consiste à diminuer la probabilité de voir des menaces se concrétiser, à en limiter les effets ou les dysfonctionnements induits et à autoriser le retour à un fonctionnement normal à des coûts et des délais acceptables en cas de sinistre.

En ce qui concerne la protection des données et des services, cela implique qu'il faille assurer les propriétés suivantes [GH00a, Mai02] : *confidentialité* (aucun accès illicite), *intégrité* (aucune falsification), *exactitude* (aucune erreur), *disponibilité* (aucun retard), *pérennité* (aucune destruction), *non-répudiation* (aucune contestation).

Ces propriétés doivent être garanties par des mesures de sécurité. Celles-ci sont mises en oeuvre au travers d'outils particuliers, de procédures adaptées et de personnes formées ou sensibilisées à la sécurité. Elles sont gérées et validées par des procédures de gestion et d'audit.

La mission de la sécurité peut donc se résumer en actions génériques qui consiste à [GH00a] :
- définir le périmètre de la vulnérabilité lié à l'usage des technologies de l'information et de la communication,
- offrir un niveau de protection adapté aux risques encourus par l'entreprise,
- mettre en oeuvre et valider l'organisation, les mesures, les outils et les procédures de sécurité,
- optimiser la performance du système d'information en fonction du niveau de sécurité requis,
- assurer les conditions d'évolution du système d'information et de son schéma de sécurité associé.

1.4.2 Les capacités d'un système d'information

La sécurité d'un système d'information vise principalement à préserver les aptitudes minimales (i.e. capacités) qu'un utilisateur est en droit d'attendre de celui ci [GH01, Mai02] :

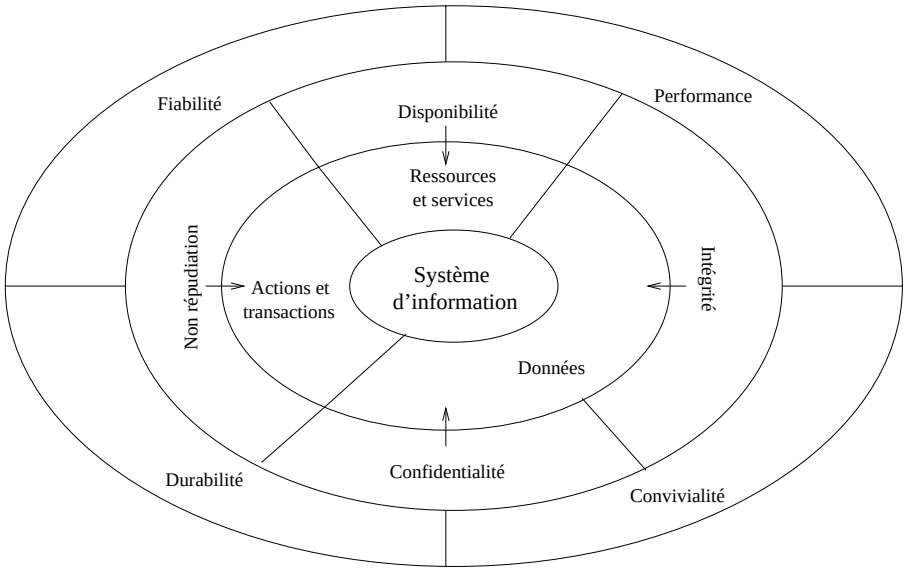


FIG. 1.11 – La sécurité d'un système d'information [GH01].

- *Capacité d'un système à pouvoir être utilisé* - elle dépend de la ***disponibilité*** des ressources et des services du système d'information dans des conditions de performance et d'utilisation adaptée (i.e. sûreté de fonctionnement).
- *Capacité d'un système à ne permettre l'accès aux données qu'aux personnes et processus autorisés* - il s'agit de préserver la ***confidentialité*** des données ; confidentialité des informations privées et personnelles à un individu, confidentialité des informations stratégiques qui ne doit pas être connues du grand public. Les données sont protégées par les processus de contrôle d'accès (Chapitre 2) et par les mécanismes de cryptographie.
- *Capacité d'un système à permettre la modification des données uniquement par les personnes ou processus habilités* - c'est l'intégrité des données qui est réalisée par les procédures de contrôle d'accès, de contrôle d'erreur ou de contrôle de cohérence.
- *Capacité d'un système à prouver que des actions, transactions ont bien eu lieu*, ceci à des fins de traçabilité, de preuve, de contrôle, d'audit ou de non répudiation d'actions.
- *Capacité d'un système à exécuter les actions et à rendre les services que l'on attend de lui* dans des conditions de performances et d'utilisation adaptées tout au long de sa vie ; cela traduit un besoin de continuité, de durabilité, de fiabilité, de convivialité et de sûreté de fonctionnement.

Ces capacités définissent les services que doit offrir, par la mise en oeuvre de moyens adaptés, un système sécurisé (Figure 1.11). Les moyens pour y parvenir sont rappelés dans le Tableau 1.2.

1.4.3 Les typologies des risques

Dans un premier temps, on peut analyser les risques selon la nature de leurs impacts en termes de disponibilité, d'intégrité, de confidentialité ou de non répudiation, c'est-à-dire sur les capacités centrales du système d'information (premier niveau de capacités de la Figure 1.11).

On peut aussi les classer, dans un second temps selon leur origine :

1. *accidents* - elles sont principalement d'origine matérielle (pannes, crashes dysfonctionnements des matériels), logicielle (bugs) ou liées aux catastrophes naturelles (inondations, incendies, ...),

Capacité d'un système à :	Critères de sécurité	Moyens de sécurité
Pouvoir être utilisé	Disponibilité	Dimensionnement Redondance Procédures d'exploitation et de sauvegarde
Exécuter des actions	Sécurité de fonctionnement Fiabilité Durabilité Continuité	Conception Performances Ergonomie Qualité de service Maintenance opérationnel
Permettre l'accès aux données autorisées	Confidentalité Intégrité	Controle d'accès Controle d'erreur Controle de cohérence Chiffrement
Prouver des actions	Non-répudiation	Certification Enregistrement et traçabilité Signature éléctronique Mécanismes de preuve

TAB. 1.2 – Les capacités d'un système, les critères et les moyens de sécurité associés.

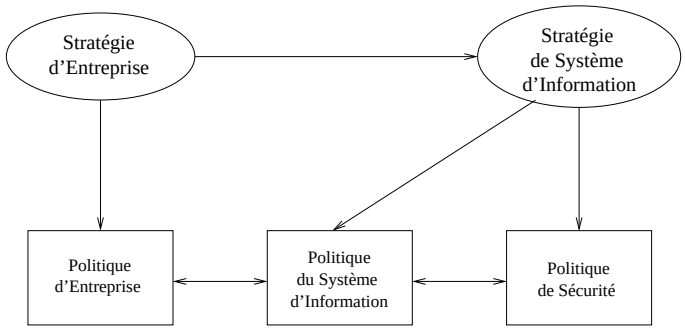


FIG. 1.12 – De la stratégie d'une entreprise à la politique de sécurité.

2. *erreurs* - erreurs de saisie, de transmission et d'utilisation des informations, erreurs d'exploitation, erreurs de conception et de réalisation des logiciels,
3. *malveillances* - vol, sabotage de matériel, fraude, sabotage immatériel (détournement d'usage, détournement de biens), indiscrétions, détournements d'informations, détournement de logiciel; elles se caractérisent par une lecture non autorisée (piratage), une modification ou une destruction non autorisée d'informations. On parle dans ce cas d'attaques.

1.4.4 Stratégies de sécurité

La sécurité informatique d'une entreprise doit s'appréhender d'une manière globale par la définition d'une stratégie ou politique de sécurité.

Une ***politique de sécurité*** est un ensemble de règles et de directives élaborées en fonction d'une analyse de risques pour en diminuer la survenue et pallier à leur apparition (Figure 1.12).

La définition d'une stratégie de sécurité doit aboutir à la mise en oeuvre de mesures concrètes. Plusieurs niveaux d'intervention sont généralement admis :

1. face aux menaces potentielles identifiées il s'agit de prendre :

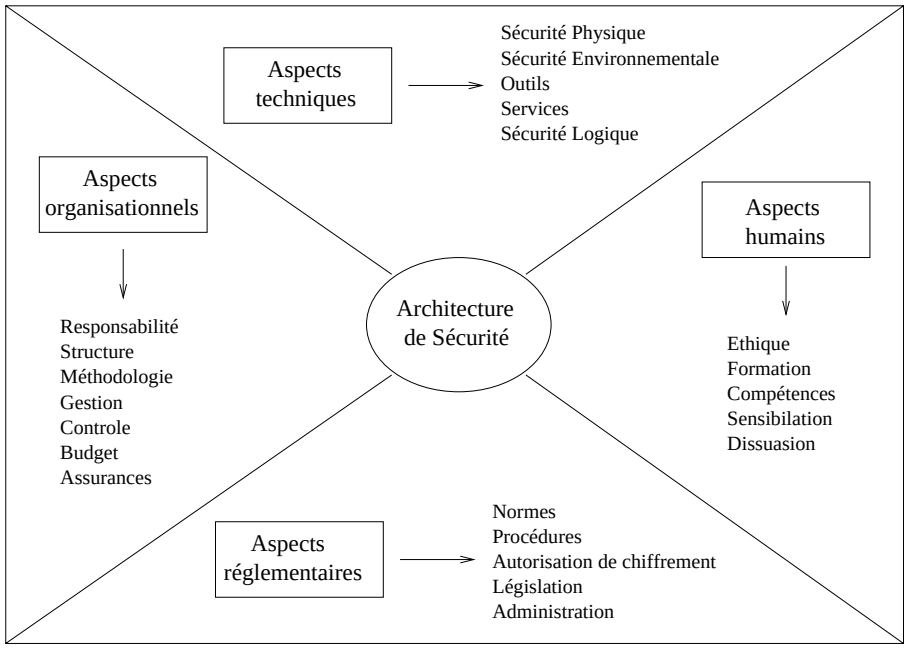


FIG. 1.13 – Les différents aspects d'une architecture de sécurité [GH01].

(a) des *mesures préventives* (contrôle d'accès physique et logique, détection de virus, etc.) pour empêcher l'aboutissement d'une menace (intentionnelle ou non) ou du moins d'en limiter l'ampleur à l'aide de plans de sauvegardes,

(b) des *mesures dissuasives* (procédures juridiques, sensibilisation des personnes, moyens de détection et de traçabilité, etc.) pour décourager les intrus éventuels,

(c) des *mesures structurelles* (occultation des ressources, redondances, fragmentation de l'information, etc.) pour protéger les biens de l'entreprise,

2. face aux attaques abouties des *mesures correctives* (plans de secours) pallient ou réparent les dégâts engendrés et des *mesures de récupération* (assurances, actions en justices) limitent les pertes financières de l'entreprise.

1.4.5 Une architecture de sécurité

La sécurité informatique d'une entreprise passe par la motivation du personnel, la mise en place de mesures ainsi que par l'utilisation d'outils. L'architecture de sécurité d'un système d'information permet de mieux visualiser globalement la sécurité. Les diverses composantes de cette architecture doivent être développées de façon cohérente, complémentaire et harmonieuse.

La Figure 1.13 présente les différentes facettes organisationnelles, humaines, réglementaires et techniques intervenant dans la maîtrise de la sécurité des systèmes d'information.

Tous les niveaux de l'informatique sont concernés par la sécurité d'un système d'information. On distingue :

1. **sécurité physique** - concerne les aspects liés à la maîtrise des systèmes (matériels, composants, etc.) et de l'environnement dans lequel ils se situent (locaux, alimentation, énergétique, climatisation, etc.),
2. **sécurité de l'exploitation** - tout ce qui touche au bon fonctionnement des systèmes d'information ; cela comprend la mise en place, l'exploitation, la maintenance et la mise-à-jour,

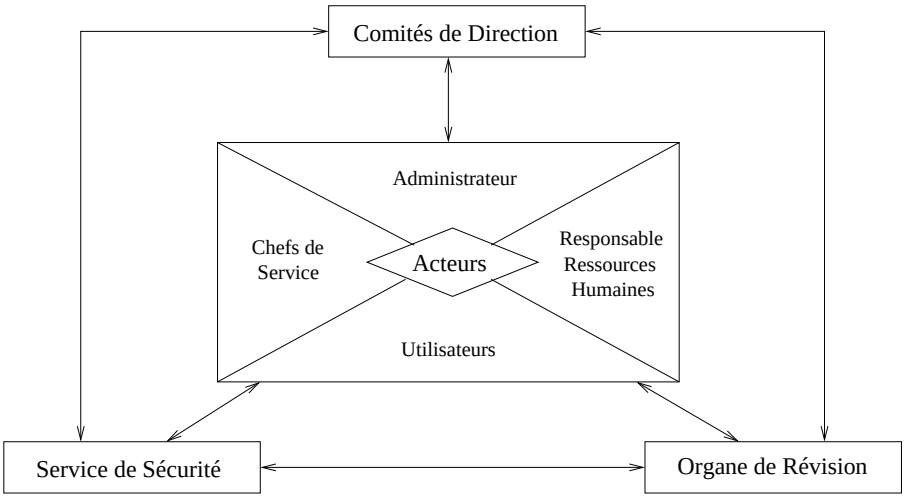


FIG. 1.14 – La structure organisationnelle de la sécurité.

3. **sécurité logique** - un ensemble de mécanismes de sécurité de type logiciel ; elle repose sur un processus de contrôle d'accès basé sur trois activités : identification, authentification et autorisation. Elle permet d'assurer la confidentialité et l'intégrité des données,
4. **sécurité applicative** - il s'agit de développer méthodologiquement de nouvelles applications robustes et sécurisées et de les intégrer harmonieusement dans les environnements opérationnels,
5. **sécurité des télécommunications** - consiste à offrir aux applications communicantes, une connectivité de bout en bout fiable et de bonne qualité.

1.4.6 La structure organisationnelle de la sécurité

La gestion de la sécurité est spécifique à la structure organisationnelle de l'entreprise et dépend de sa stratégie. Il existe donc autant de politiques, de procédures, d'outils de sécurité que d'entreprise et de besoins de sécurité.

La gestion de la sécurité sera facilitée si elle s'inscrit dans le cadre d'une démarche méthodologique de projet avec une forte implication directoriale. La direction générale de l'entreprise est responsable de l'évaluation des risques, de l'établissement de la politique de sécurité et de la mise en place de la structure organisationnelle qui la mettra en oeuvre.

L'entreprise doit créer une structure de sécurité qui, pour assurer sa mission correctement, doit être une entité autonome, indépendante des unités opérationnelles ou fonctionnelles de l'entreprise. C'est son conseil d'administration qui accordera le budget de celle-ci. Il a la responsabilité de la haute surveillance sur les personnes qui seront chargées de la gestion. La Figure 1.14 présente la structure organisationnelle de la sécurité.

La structure organisationnelle de la sécurité contient les éléments suivants [GH00a, GH01] :

- le *comité de direction* - il est chargé d'entériner la politique de sécurité, de désigner les personnes et leur assigner leur responsabilités, de valider les plans stratégiques, tactiques, opérationnels et l'audit de la sécurité,
- le *service de sécurité* - pour définir le programme de sécurité, identifier les cibles de la sécurité, élaborer et documenter la politique de sécurité, évaluer et sélectionner les composants dédiés à la sécurité,
- l'*organe de révision* - pour établir un plan de sécurité de l'existant, les différentes cibles de sécurité font l'objet d'un audit. L'organe de révision doit être externe et mandaté par le conseil d'administration,

- les *acteurs* - chaque acteur de l'entreprise doit être actif dans le dispositif :
  - un *administrateur de la sécurité* - une personne en contact avec la cible sécuritaire ; il est responsable de son maintien en condition d'exploitation ; il effectue les tâches suivantes :
    - gestion et administration des moyens et des mesures de sécurité nécessaires à la cible dont il est responsable,
    - définition des privilèges de chacun d'accès aux ressources du système,
    - réévaluation régulière des privilèges d'accès suite aux évolutions et aux changements des besoins,
    - surveillance de la cible de sécurité en analysant tout incident de nature à compromettre la sécurité,
  - le *responsable des ressources humaines* - il est chargé de l'engagement de nouveaux collaborateurs et de les sensibiliser face à leur future responsabilité en matière de sécurité,
  - les *chefs de service* - veillent à ce que leurs collaborateurs respectent les consignes de sécurité, informent les administrateurs de toute modification ayant un impact sur la gestion de la sécurité,
  - les *utilisateurs* - ils doivent s'engager à respecter les règles et les pratiques définies par la politique de sécurité. Ils doivent utiliser les ressources informatiques de l'entreprise mises à leur disposition uniquement dans le cadre de leur fonctions. C'est généralement le maillon faible d'une architecture de sécurité.

1.5 Les méthodes de conception pour la sécurité des systèmes d'information

La sécurité des systèmes d'information a longtemps consisté à mettre en oeuvre des solutions génériques sur les systèmes et les réseaux de l'entreprise. Il suffisait de détecter des vulnérabilités pour mettre en évidence des risques et pour décider des mesures les plus appropriées.

Les méthodes de management de la sécurité des années 80 et 90 ont été basées sur ce principe.

Il est souhaitable qu'une gestion du système d'information soit conduit de façon méthodologique et que celle-ci corresponde à une norme réelle ou à un standard de fait, afin de pouvoir comparer des solutions selon les mêmes critères.

On peut citer quelques méthodes utilisées en France : *Marion* (la méthode publique), *Melisa* (la méthode semi-privée) ou *Méhari*.

1.5.1 La présentation de la méthode Marion

Depuis son introduction en France en 1984, la méthode **Marion** (*Méthode d'Analyse des Risques Informatiques et d'Optimalisation par Niveau*) a été utilisée dans d'autres États d'Europe occidentale et au Québec. La méthode est naturellement très populaire en France, d'où elle est originaire. Elle est appliquée par quantité de banques, de mutuelles, de groupes industriels et de petites et moyennes entreprises.

Marion est un ensemble méthodologique conçu et développé par le CLUSIF (Club de la Sécurité des Systèmes d'Information Français), qui réunit la plupart des spécialistes français des systèmes d'information. Elle a été adaptée à divers contextes (grands, moyens et petits systèmes) et traduite dans la plupart des pays à économie développée.

Elle est suffisante dans de nombreux cas et elle permet en quelques jours de faire une "photographie" à un instant donné de l'état de sécurité d'un système d'information.

Sous la forme la plus générale, pour les grandes entreprises (multi-sites et multi-systèmes), Marion est composée de trois volets schéma directeur sécurité des systèmes d'information (SDSSI) : le *schéma préalable* suivi des *schémas locaux* (correspondant aux sites informatiques), eux-mêmes consolidés par le *schéma de synthèse* (Figure 1.15).

Chaque volet de SDSSI est supporté par un module méthodologique de Marion. Les grands principes de base de Marion sont les suivants [Lam91] :

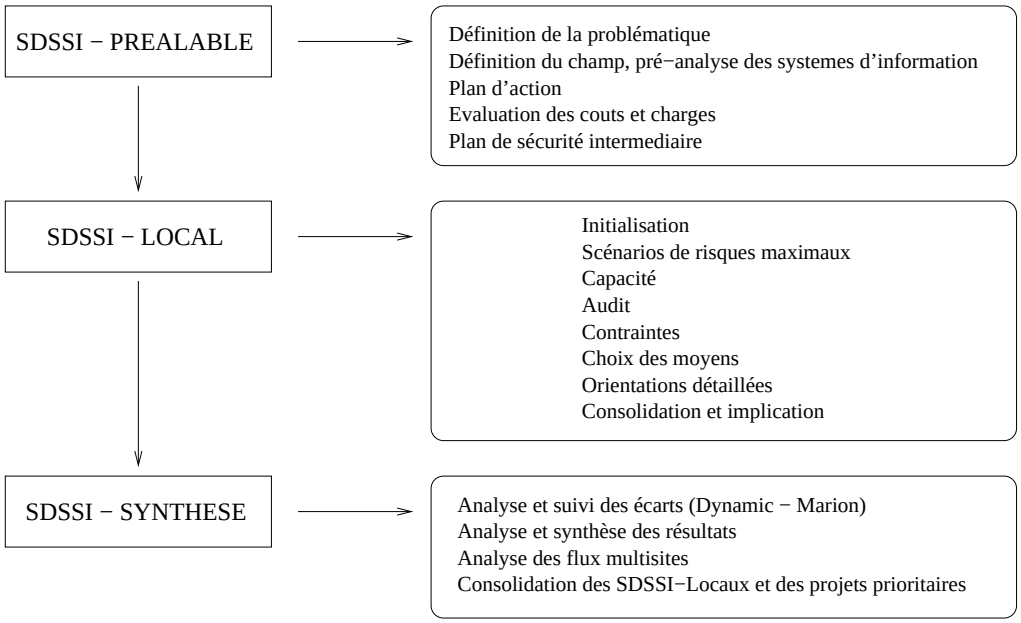


FIG. 1.15 – Les trois schémas de Marion.

- la sécurité ne peut être assurée que s'il y a une forte implication de l'entreprise ou de l'organisme de la direction générale à l'utilisateur final,
- l'approche doit être pluridisciplinaire et participative - la mise en oeuvre de moyens et procédures de sécurité réalistes et cohérentes entre eux, ne peut dépendre d'un responsable de sécurité isolé mais d'un comité de pilotage comprenant des informaticiens, des utilisateurs, des organisateurs, etc.,
- le schéma directeur de sécurité des systèmes d'information est une base de décision pour la direction générale : toutes les informations doivent y être objectivement quantifiées pour justifier le budget de sécurité par rapport aux objectifs du plan d'entreprise,
- la méthode est normative et contient dans chaque étape des supports de guidage (outils et logiciels),
- la méthode doit être suffisamment globale et prendre en compte les questions de sécurité et de qualité des systèmes d'information ; elle vise à réduire les vulnérabilités en cas d'accidents, les erreurs et les malveillances afin d'assurer la sécurité dans les domaines de la disponibilité, de l'intégrité et de la confidentialité ; les règles de base sont : la cohérence des moyens de sécurité et leur adéquation aux enjeux.

Les étapes de base de la méthode Marion sont les suivantes :
- *analyse des risques majeurs* - permet de déterminer exhaustivement et quantitativement les scénarios de risques majeurs courus par l'entreprise,
- *expression du risque maximum admissible* - permet de calculer la capacité de l'entreprise, c'est-à-dire la perte maximale qu'elle pourrait subir du fait des événements redoutés sans mettre en péril sa survie,
- *analyse des moyens de la sécurité* - cette étape est fondée sur un audit qui permet d'inventorier et de qualifier de manière cohérente et exhaustive les moyens de la sécurité,
- *évaluation des contraintes* - cette étape introduit la notion de contrainte d'équilibrage prévention/protection qui permet de préciser la part minimale du budget à allouer à la prévention afin de minimiser le coût des sinistres potentiels,
- *choix des moyens* - liste les moyens à mettre en oeuvre ou à améliorer,
- *orientation et rapport* - permet de définir le plan technique détaillé ; on précise la structure et l'organisation nécessaires à l'application, au suivi et à la révision ultérieure du SDSSI ; on

rédige enfin le SDSSI sous la forme normative.

Marion constitue un standard de fait pour l'évaluation de la sécurité des systèmes d'information. Elle permet aussi de spécifier le contrôle d'accès logique aux données d'un système d'information. La définition des règles générales du contrôle d'accès doit être réalisée au niveau du schéma préalable dont on peut déterminer la politique globale d'une entreprise. Le niveau du schéma local correspondant aux sites locaux peut cadrer les règles générales du contrôle d'accès aux besoins locaux. Ce niveau permet de faire des modifications de la politique du contrôle de niveau général en d'accord avec les exigences des utilisateurs locaux d'une organisation. Il permet aussi de choisir des moyens pour mettre en oeuvre les règles définies du contrôle d'accès.

La méthode est applicable à des organisations de tailles diverses utilisant différentes plates-formes et technologies (systèmes centraux, micro-informatique, réseaux locaux, réseaux étendus). Elle est évolutive ; un groupe permanent du CLUSIF, la commission Méthodes, en est responsable.

Cette méthode mène à l'équilibrage des actions de protection et de prévention. Elle peut être adaptée à tout contexte et chacun de ses volets peut être utilisé indépendamment.

1.5.2 La présentation de la méthode Méhari

La méthode **Méhari** (*MEthode Harmonisée d'Analyse de RIsques*) résulte des travaux de Jean-Philippe Jouas et de Albert Harari et de leur consolidation au sein de la commission Méthodes du Clusif. Ils ont publié les résultats de leurs travaux en 1992.

A. Harari avait développé Melisa pour la DCN (Direction des Constructions Navales de la Délégation Générale pour l'Armement). Cette méthode proposait une certaine vision des risques, avec des paramètres d'évaluation relativement simples et adaptés à la cible visée. Sur la base de ce modèle et en incorporant le savoir-faire et les bases de connaissances acquis par le Clusif avec la méthode Marion depuis 1984, s'est construit progressivement un ensemble d'outils de management de la sécurité que l'on a appelé Méhari.

1.5.2.1 Les objectifs de Méhari

La méthode a pour objectif de répondre aux demandes suivantes :
- avoir une approche globale de la sécurité dans des structures décentralisées,
- gérer la complexité de la sécurité d'un système d'information,
- permettre une approche analytique dans l'évaluation des risques et la recherche de solution.

Une caractéristique essentielle de la méthode Méhari est de permettre l'évaluation réaliste des risques et également le contrôle et la gestion de la sécurité de l'entreprise sur le court, le moyen et le long terme.

La méthode Méhari permet de concilier les objectifs stratégiques et les nouveaux modes de fonctionnement de l'entreprise avec une politique de maintien des risques à un niveau convenu. L'analyse des vulnérabilités fait partie de la méthode en soutien du management des risques.

1.5.2.2 La méthodologie de Méhari

La méthode Méhari s'articule autour de trois phases (Figure 1.16) :

1. le *Plan Stratégique de Sécurité (PSS)* qui fixe les objectifs de sécurité et les métriques associées, qualifie le niveau de gravité des risques encourus,
2. les *Plans Opérationnels de Sécurité (POS)* qui déterminent, par site ou entité géographique, les mesures de sécurité à mettre en place, tout en assurant la cohérence des actions choisies,
3. le *Plan Opérationnel d'Entreprise (POE)* qui permet le pilotage de la sécurité au niveau stratégique par la mise en place d'indicateurs et la remontée d'information sur les scénarios les plus critiques.

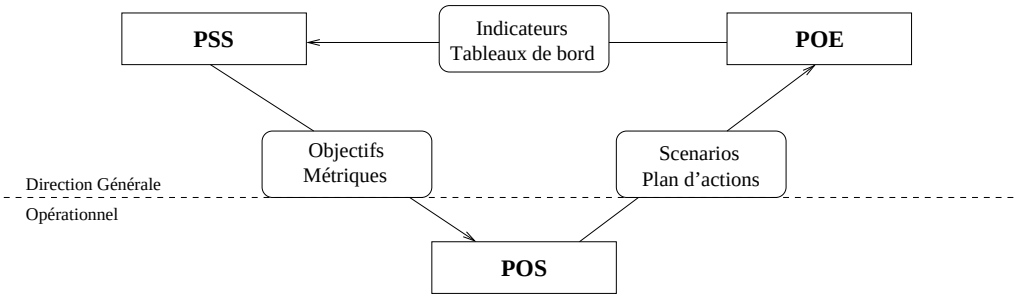


FIG. 1.16 – La démarche Méhari.

1.5.2.3 Les éléments de Méhari

L'ensemble méthodologique de Méhari comprend les éléments suivants [Clu] (Figure 1.17) :
- un ensemble de *guides* à destination des managers et du Responsables de la Sécurité de Systèmes d'Information (RSSI), qui facilitent :
  - la classification des ressources sensibles du système d'information,
  - l'audit des services de sécurité,
  - l'analyse des risques,
  - l'élaboration d'objectifs de sécurité,
- un ensemble de *bases de connaissances* qui :
  - décrit l'ensemble des services de sécurité. Les services sont regroupés par domaines tels que la sécurité physique des sites, l'organisation générale, le *contrôle d'accès logique*, etc.
  - regroupe des questions pertinentes quant à la qualité des services de sécurité,
  - contient un ensemble de scénarios de risque classés par familles de conséquences.
  - indique, pour chaque scénario, quels sont les services de sécurité qui peuvent avoir un effet sur le niveau de risque résultant et, dans ce cas, quel effet.
  - permet l'analyse des scénarios de risque par une évaluation directe des facteurs de risque.
- des *processus d'évaluation quantitative* qui permettent de :
  - fournir une évaluation globale de chaque service sous la forme d'une note de qualité de service,
  - fournir une évaluation des facteurs de risque pour chaque scénario,
  - fournir une évaluation de la potentialité et de l'impact d'un scénario,
  - passer des évaluations de la potentialité et de l'impact à une évaluation globale de la gravité du risque.

Les guides et les bases de connaissance sont décrits dans la documentation Méhari fournie par le Clusif. Les moteurs sont fournis en standard sous forme d'algorithme par le Clusif.

Les retours d'expériences ont permis de mieux adapter le questionnaire d'audit et d'y incorporer l'étude des risques lies à Internet ainsi qu'aux Nouvelles Technologies de l'Information et des Communications (NTIC) et à l'externalisation des moyens informatiques.

1.6 Conclusion

Le choix d'une méthode pour la conception des systèmes d'information et de ses systèmes de sécurité dépend de différentes conditions : le type d'entreprise, le type du futur système d'information, les besoins des futurs utilisateurs du système, des préférences du concepteur et des possibilités de la méthode.

Nous avons présenté deux méthodes de conception et de développement des systèmes d'information, Merise et Ossad. Ces méthodes possèdent des éléments qui permettent de modéliser

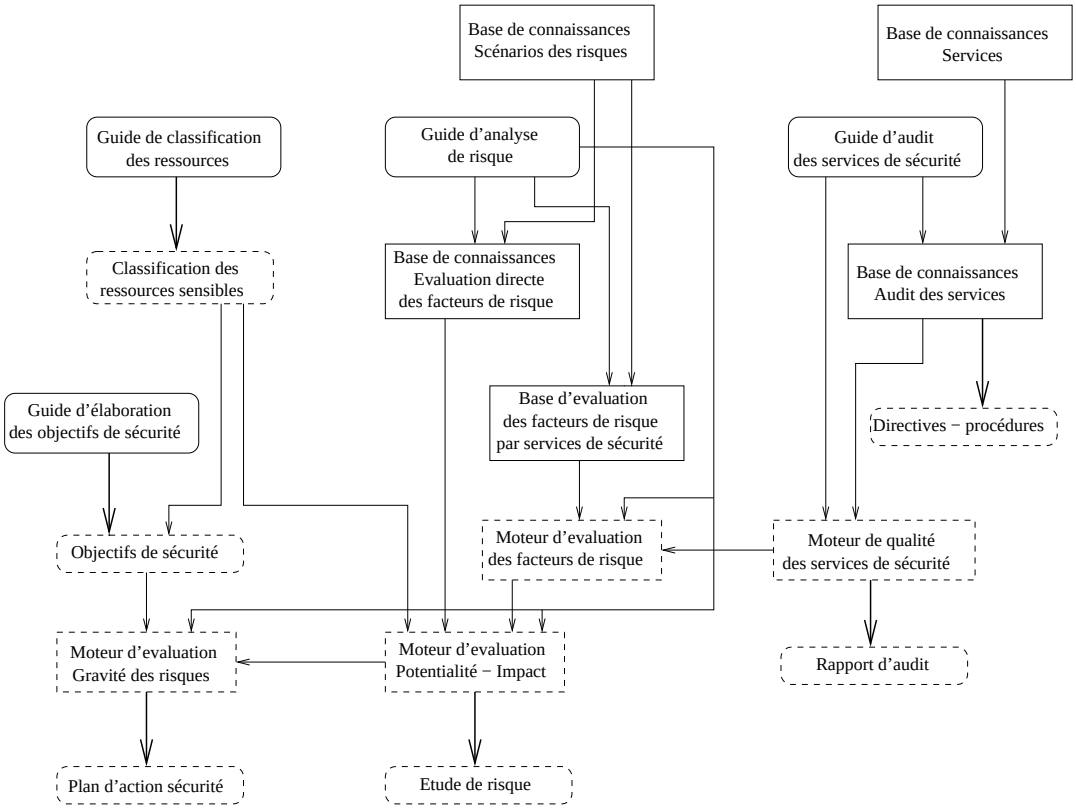


FIG. 1.17 – La démarche d'élaboration de plans de sécurité de la méthode Méhari.

l'organisation de l'entreprise, en particulier de préciser l'accès aux données (Merise) ou d'introduire la notion de rôle (Ossad).

La gestion de la sécurité est spécifique à la structure de l'entreprise et dépend de sa stratégie générale. Cette dernière prend en compte les objectifs généraux de l'entreprise en termes de marketing de production et d'organisation. Une politique de sécurité offre une réponse à un problème sécuritaire spécifique, en fonction d'une analyse des risques potentiels qui pèsent sur une organisation et d'un compromis financier.

La partie suivante de ce chapitre a montré des méthodes dédiées pour la sécurité des systèmes d'information. Défendre les informations contre des révélations impropres ou modifications (i.e. capacités de confidentialité et d'intégrité) est une exigence importante de chaque système d'information. C'est dans ce cadre particulier du contrôle d'accès aux informations et aux services du système d'information que vont se situer nos travaux.

Le prochain chapitre présente les différentes politiques de sécurité pour contrôler les accès aux données ou services du système d'information.

La mise en place d'un contrôle d'accès devra répondre aux trois questions suivantes :

- Qui peut faire quoi ?
- Comment vérifier ?
- Comment protéger ?

Chapitre 2

Le contrôle d'accès dans les systèmes d'information

Objectifs du chapitre :
- *Définir le contrôle d'accès dans un système d'information*
- *Décrire les différentes politiques de contrôle d'accès et leurs modèles*
- *Présenter un modèle basé sur la notion de rôle - le modèle RBAC*
- *Examiner les travaux de recherche portant sur les méthodologies centrées sur l'ingénierie des rôles*

Défendre les informations contre des révélations impropre ou modifications est une exigence importante de chaque système d'information.

Dans ce cadre, le contrôle d'accès permet de limiter l'activité d'un utilisateur légal aux fonctionnalités du système d'information qui lui sont nécessaires pour assurer sa mission dans l'entreprise. Le contrôle d'accès fait partie intégrante de la sécurité logique à mettre en place dans tout système d'information car elle repose sur des mécanismes de sécurité de type logiciel. Le contrôle d'accès est généralement lié avec un service d'audit qui permet l'enregistrement dans un journal de toutes les opérations effectuées par chaque utilisateur sur le système d'information. Une analyse ultérieure de ce journal permettra de détecter des intrus du système d'information ou de mettre en place des procédures de recouvrement [AA92, SJ93, San94, TS94].

Le contrôle d'accès d'un système d'information doit permettre l'accès aux entités autorisées tout en garantissant la confidentialité et l'intégrité des objets du système d'information. Elle suppose la mise en oeuvre d'un processus de contrôle d'accès logique qui s'appuie sur trois services complémentaires :
- un service d'**identification** qui permet d'identifier de manière unique chaque utilisateur par un nom de login,
- un service d'**authentification** qui permet de valider de manière sûre chaque utilisateur qui s'identifie,
- un service de **gestion des autorisations** qui indique les opérations qu'un utilisateur légal (i.e. authentifié avec succès) peut effectuer sur le système d'information.

Ce chapitre présente les mécanismes et les politiques pour gérer le contrôle d'accès. La première partie définit le contrôle d'accès aux informations et résume les différentes politiques de contrôle d'accès. La deuxième partie décrit les modèles de contrôle d'accès de type DAC et de type MAC. Ensuite, on présente dans une troisième partie un modèle basé sur la notion de rôle - le modèle RBAC. Dans la dernière partie, nous examinons les travaux de recherche portant sur des méthodologies centrées sur l'ingénierie des rôles.

2.1 L'accès aux informations

Une menace est définie comme un agent, qui en utilisant une technique spéciale, effectue une action visant à violer les propriétés du système d'information.

Une action malveillante peut se traduire par :
- une lecture, une modification ou une suppression d'information effectuées par une personne non autorisée,
- l'obtention impropre d'une information par une personne qui abuse de ses privilèges.

Dans le premier cas, il s'agit de se prémunir des ***accès directs*** non autorisés alors que dans le second cas il faut se prémunir des ***accès indirects***.

Pour contrôler les accès directs, un mécanisme de ***contrôle d'accès*** doit être mis en place. Celui ci sera décrit plus particulièrement dans le paragraphe suivant.

Pour maîtriser les accès indirects aux informations, des mécanismes de ***contrôle de flux*** et de ***contrôle d'inférence*** empêchent respectivement la divulgation d'information et la déduction d'information obtenue à partir de récoupages multiples.

Notons enfin, que lorsque la mise en place d'un contrôle n'est pas possible, le ***cryptage*** des données permet de rendre les informations inintelligibles pour les utilisateurs ne disposant pas des clés de décryptage.

2.1.1 Le mécanisme d'authentification

Le principe est de permettre une identification correcte d'un utilisateur. Pour obtenir ce but, les mécanismes d'authentification valident l'identité d'un utilisateur en utilisant certains objets ou/et informations, détenus ou connus pour l'utilisateur.

Les systèmes d'authentification basés sur des informations connues par un utilisateur sont les suivants [CFMS94, SS94, SCFY94] :

1. *systèmes basés sur le mot de passe* - un utilisateur est identifié par un mot de passe secret connu exclusivement par cet utilisateur et le système,
2. *systèmes basés sur une question-réponse* - un utilisateur est identifié sur la base de ses réponses à un ensemble de questions spécifiques à celui-ci qui lui sera posé par le système.

Les systèmes d'authentification basés sur des informations possédées par l'utilisateur reposent souvent sur l'utilisation d'une carte magnétique ou à puce. Les systèmes d'authentification basés sur les caractéristiques propres à l'utilisateur procèdent généralement par comparaison ou par reconnaissance avec un modèle de référence stocké dans le système.

Les mécanismes basées sur un mot de passe sont les plus couramment utilisés car les moins coûteux à mettre en oeuvre. Ce sont malheureusement les moins sûrs car il faut les choisir compliqués et les changer fréquemment, ce qui n'est pas souvent du goût des utilisateurs. Une fois l'utilisateur authentifié avec succès, on parle alors d'utilisateur légal ou légitime.

2.1.2 Le contrôle d'accès

Le but du ***contrôle d'accès*** est la limitation des actions ou des opérations qu'un utilisateur légitime d'un système information peut exécuter sur celui-ci. Il s'agit donc de contrôler les *accès directs* d'un utilisateur aux informations du système d'information. De cette façon le contrôle d'accès est classé parmi les moyens de prévention pour sécuriser un système d'information [CFMS94].

Le contrôle d'accès dans un système d'information doit vérifier que tous les accès directs aux objets du système se font en accord avec les modes et les règles fixés par les politiques de protection choisies lors de la phase de conception. Le contrôle d'accès doit permettre l'accès aux entités autorisées afin de garantir la confidentialité et l'intégrité des objets du système d'information.

Le contrôle d'accès consiste à vérifier la validité de l'opération suivante :

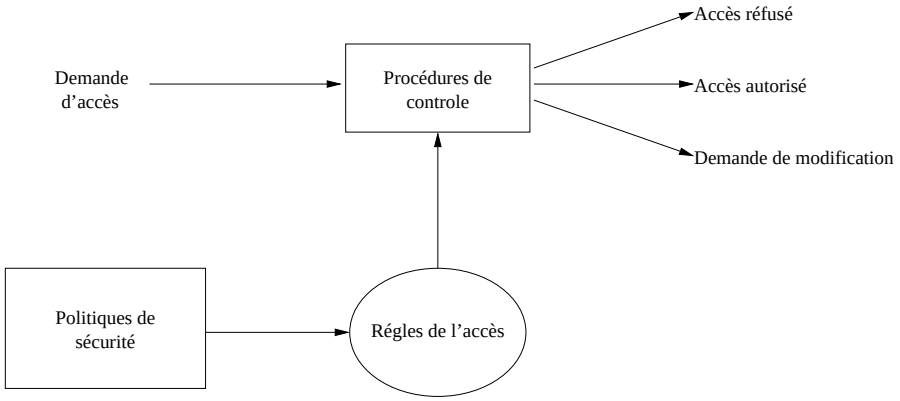


FIG. 2.1 – Un système de contrôle d'accès

"Un sujet (utilisateur, processus) accède aux informations (données, programmes) en effectuant des opérations (lecture, écriture, suppression, exécution) qui respectent les règles d'accès ".

Il contient deux composantes (Figure 2.1) :

1. Un ensemble de politiques de sécurité et de règles d'accès - les règles indiquent les modes d'accès possibles que les sujets peuvent effectuer sur les objets. Les politiques indiquent comment ces règles sont définies et vont évoluer.
2. Un ensemble de procédures de contrôle (les mécanismes de sécurité) qui valident ou non les requêtes (les demandes d'accès) en se basant sur les règles établies ; la demande peut être accordée, refusée ou demandée à être modifiée.

2.1.2.1 Les concepts d'une politique de contrôle d'accès

Pour définir une politique de contrôle d'accès, on utilise les concepts suivants :

- **objets** - un objet est une entité passive du système, qui contient l'information à protéger contre les accès ou modifications non autorisés. Celui-ci peut être un fichier, un enregistrement dans un fichier ou bien un champ particulier d'un enregistrement,
- **sujets** - un sujet est une entité active du système (i.e. utilisateur ou processus), qui effectue des requêtes d'accès aux objets. Ces requêtes doivent être contrôlées pour s'assurer qu'elles ne portent pas atteinte à la sécurité du système. Tout sujet représente une menace potentielle (i.e. intentionnelle ou accidentelle) à la sécurité de l'information,
- **modes d'accès** - le type d'accès (lecture, écriture, etc.) que les sujets peuvent effectuer sur les objets. L'exécution d'un mode d'accès produit un transfert de flux d'information de l'objet vers le sujet,
- **politiques** - définissent les principes sur lesquels reposent la gestion et la conception des autorisations,
- **autorisations** - elles définissent les accès que les sujets peuvent effectuer sur les objets du système. L'état d'autorisation d'un système est constitué par l'ensemble des autorisations. On distingue deux types d'autorisations : les positives qui accordent une permission d'accès et les négatives qui interdissent un accès,
- **droits d'administration** - ces privilèges (implémentés par des primitives comme "grant", "revoke") permettent la modification des autorisations. Dans un système discrétionnaire la possession des droits administratifs par un sujet lui confère le droit de donner ou de retirer des autorisations à d'autres sujets (i.e. par délégation),
- **axiomes** - des propriétés qui doivent être satisfaites dans le système. Elles indiquent les conditions à remplir par l'état d'autorisation du système. La modification de cet état ne peut se faire que si l'état modifié satisfait également les axiomes. La satisfaction des axiomes assure que le système est sécurisé en respectant la politique de protection choisie.

La prochaine section présente les éléments permettant de définir différentes politiques de contrôle d'accès.

2.1.2.2 Les politiques de contrôle d'accès

Les ***politiques*** de sécurité définissent les grands principes sur lesquels reposent la gestion et la conception des autorisations. Il s'agit de déterminer les droits initiaux de chacun et la manière dont ceux-ci peuvent évoluer. Les politiques de sécurité doivent aussi indiquer la manière d'administrer l'ensemble de ces autorisations (Qui et Comment).

Les ***règles d'accès*** sont les expressions des autorisations qui résultent de l'application des politiques de sécurité définies au niveau stratégique. Les règles d'accès indiquent si un **sujet s** peut ou non accéder à un **objet o** en utilisant un **mode opératoire op**.

Le choix d'une politique de sécurité dépend dans un premier temps de la fonctionnalité finale du système, c'est-à-dire de la réponse aux questions suivantes :

1. **Un système d'information pour quel usage ?**
   Les systèmes d'information peuvent être classés selon leur vocation. On distingue principalement deux catégories :
   (a) **les systèmes à vocation "grand public"**, où la plupart des données sont destinées à être partagées, favorisent ainsi la diffusion des informations. Ce type de système est fortement utilisé dans le milieu universitaire,
   (b) **les systèmes à vocation "privé"**, où la plupart des données doivent être protégées car contenant des informations à caractère confidentiel ou sensible. Ce type de système est surtout utilisé dans les milieux militaires ou industriels.
2. **Quelle quantité d'informations autoriser ?**
   On distingue pour cela principalement deux types de politiques :
   (a) **politique de privilège maximal**, qui conduit à une stratégie où chaque utilisateur dispose initialement du maximum de droits. Ce type de politique favorise le partage d'informations entre les utilisateurs et donc peut être utilisé dans les systèmes à vocation "*grand public*",
   (b) **politique de privilège minimal** ou **du moindre privilège**, qui conduit à une stratégie où chaque utilisateur dispose initialement du minimum de droits qui lui est strictement nécessaire pour son activité. Ce type de politique est utilisé dans les systèmes à vocation "*privé*".
3. **Autoriser ou interdire ?** - Après avoir choisi la politique à adopter, il est nécessaire de définir la nature des autorisations gérées par le système.
   La réponse à cette question conduit à la définition de trois systèmes :
   (a) **systèmes positifs** - où uniquement des *permissions* sont accordées (autorisations positives),
   (b) **systèmes négatifs** - où uniquement des *interdictions* sont accordées (autorisations négatives),
   (c) **systèmes mixtes** - où des interdictions et des permissions peuvent être spécifiées simultanément.
4. **Système fermé ou ouvert ?** - A ces systèmes correspondent en fait deux stratégies possibles, qui permettent la spécification par défaut des autorisations non explicitement définies :
   (a) **hypothèse du monde fermé** - tout ce qui n'est explicitement autorisé est par défaut interdit,

(b) **hypothèse du monde ouvert** - tout ce qui n'est explicitement interdit est par défaut autorisé.

Un système positif associé à un monde fermé facilite la gestion d'une politique de privilège minimal, tandis qu'un système négatif associé à un monde ouvert facilite la gestion d'une politique de privilège maximal.

5. **Qui peut accorder ou retirer des droits d'accès ?** - Après avoir déterminé le système à adopter, il reste à définir la politique de gestion des autorisations. Une permission ou une interdiction n'est pas toujours issue des prérogatives d'un seul responsable de sécurité (autorisation centralisée). Parfois la gestion des autorisations demande une certaine décentralisation ou délégation. Le choix entre une administration centralisée ou décentralisée constitue une politique de sécurité.
On distingue quatre types de gestion :

   (a) *système à autorisations centralisées* (mandatory system) - où seul un administrateur est mandaté pour effectuer ces opérations,

   (b) *système hiérarchique à autorisations décentralisées* - où l'administrateur peut déléguer ses droits à un petit groupe de personnes. L'administrateur central est responsable de la distribution des responsabilités à ses sous-administrateurs,

   (c) *système à autorisations décentralisées* (discretionnary system) - où chaque objet est administré par son propriétaire (son créateur). Un utilisateur qui a créé des objets est aussi propriétaire de ces objets. Il peut autoriser ou refuser l'accès à ses objets à d'autres utilisateurs selon son bon vouloir,

   (d) *système à autorisations coopératives* - où la modification des droits d'un sujet sur un objet nécessite l'accord unanime d'un groupe de personnes.

   Les systèmes (a) et (c) sont les plus répandus à ce jour.

Les politiques de sécurité d'un système expriment donc des choix fondamentaux qui sont pris par une organisation pour sécuriser ses données [CFMS94].

2.1.2.3 Les modèles de sécurité

Une fois les politiques de sécurité choisies, il convient de définir le modèle de sécurité à mettre en place.

L'objectif de la modélisation sécuritaire est de produire un modèle conceptuel de haut-niveau issu des exigences spécifiques qui décrivent les besoins de protection du système et indépendant du logiciel utilisé pour le matérialiser. Un ***modèle de sécurité*** fournit une représentation sémantiquement riche qui autorise la description des propriétés fonctionnelles et structurelles du système de sécurité.

Les modèles de sécurité peuvent être classés suivant les caractéristiques suivantes :

1. le *système cible* - destiné à la protection d'un système d'exploitation, d'une base de données ou les deux à la fois,
2. le *type de politique* - mandataire ou discrétionnaire,
3. la *finalité du modèle* - protection de la confidentialité, de l'intégrité ou les deux,
4. le *type de contrôle* - contrôle des flux d'accès directs ou contrôle des flux indirects (lutte contre l'inférence et les canaux cachés).

Pour faciliter la conception d'un modèle de contrôle d'accès, de nombreux modèles ont été proposés dans la littérature. On peut néanmoins les classer selon le type de politique adoptée (Figure 2.2) :

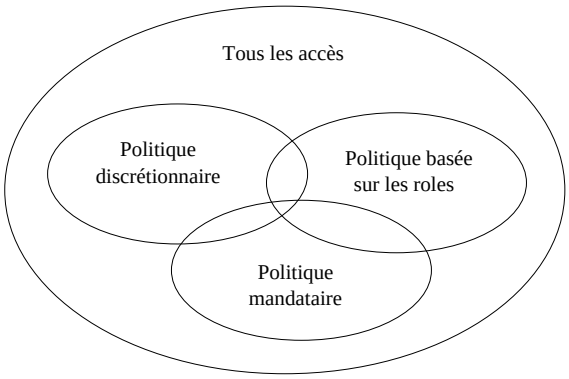


FIG. 2.2 – Les types des politiques de contrôle d'accès.

1. **contrôle d'accès discrétionnaire** (Discretionary Access Control - DAC) - des droits d'accès sont donnés pour chaque sujet,
2. **contrôle d'accès mandataire** (Mandatory Access Control - MAC) - limite les accès des sujets aux objets en se basant sur des niveaux de sécurité,
3. **contrôle d'accès basé sur les rôles** (Role-based Access Control - RBAC) - définit un rôle comme un ensemble de permissions.

Les politiques de contrôle d'accès ne sont pas mutuellement exclusives. Elles peuvent être combinées dans un même environnement où une stratégie MAC serait appliquée au niveau de la gestion des autorisations et une stratégie DAC serait appliquée au niveau du contrôle des accès.

2.2 Le modèle à politique de contrôle d'accès discrétionnaire (DAC)

Une ***politique de protection discrétionnaire*** gère l'accès des utilisateurs aux informations en se basant sur l'identité des utilisateurs et les autorisations accordées à ceux ci [TS93, CFMS94, SS94, SM98a]. Toutes les autorisations sont spécifiées pour chaque sujet et pour chaque objet du système. Il existe des règles qui spécifient les modes d'accès que ce sujet possède sur cet objet (Figure 2.3).

Le terme "discrétionnaire" indique la possibilité pour l'utilisateur d'accorder (grant) ou de retirer (revoke) à d'autres sujet les droits d'accès qu'il possède sur des objets. Ces droits ont été obtenus soit parce qu'il est propriétaire de l'objet, soit parce qu'il les a reçus d'un autre utilisateur. Ceci indique une décentralisation possible de l'administration. Mais la politique d'accès discrétionnaire est aussi bien concevable avec une gestion centralisée de celui ci.

Le système DAC possède par ailleurs des faiblesses inhérentes. La possibilité de transfert des droits a pour conséquence de permettre :
- d'une part à un droit d'accès de se propager d'un utilisateur à un autre conduisant ainsi à un éventuel transfert impropre d'information par abus de privilège,
- et d'autre part la possibilité à un cheval de Troie d'agir à l'insu d'un utilisateur digne de confiance.

Cette politique n'est pas donc adaptée pour le contrôle de flux de données. La raison est que cette politique n'impose pas de restrictions sur l'utilisation des informations par un utilisateur, la dissémination n'est pas contrôlée. Ce problème est résolu par un contrôle d'accès mandataire qui

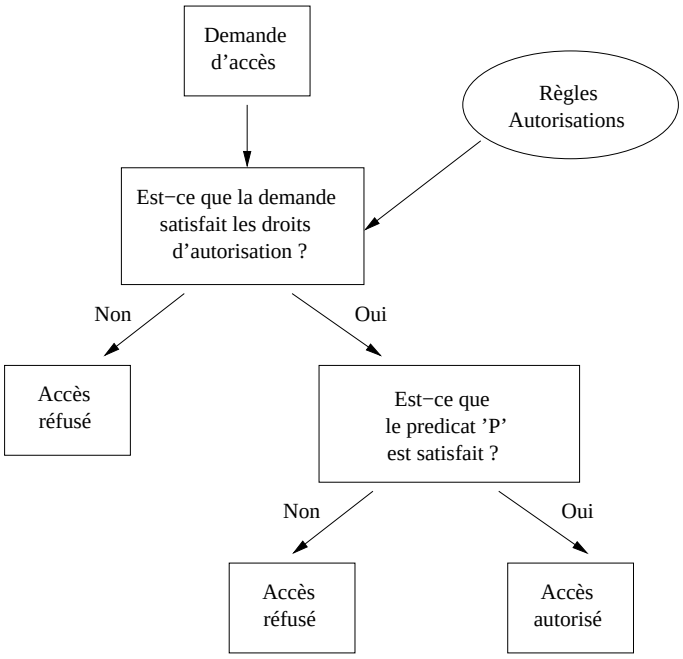


FIG. 2.3 – Le contrôle d'accès discrétionnaire.

met en place un mécanisme de contrôle de flux d'informations des objets de niveaux supérieurs vers les objets de niveaux inférieurs.

La révocation des droits obtenus par propagation est un autre problème propre à ce type de système qui nécessite pour le résoudre une gestion complexe d'un graphe de permissions.

Le contrôle d'accès discrétionnaire représente donc une solution simple et flexible pour répondre à des exigences variées de protections.

La notion de matrice d'accès, introduite par Lampson [Lam71] a inspiré de nombreux modèles de type DAC. On peut citer quelques modèles : la matrice de contrôle d'accès [HRU76, CFMS94], le modèle de Take-Grant [JLS76] ou le modèle d'Action-Entity [FM84]. Nous présentons le modèle matrice d'accès dans la prochaine section.

2.2.1 Le modèle à matrice de contrôle d'accès (ou plus simplement modèle matrice d'accès)

Le modèle à ***matrice de contrôle d'accès*** était initialement destiné à la protection d'un système d'exploitation ou d'une base de données. Il représente la référence en ce qui concerne la sécurité discrétionnaire. La plupart des modèles de sécurité peuvent être considérés comme une extension de ce modèle.

C'est un modèle conceptuel qui spécifie les privilèges que chaque sujet possède sur chaque objet au moyen d'une matrice. Il y a une ligne dans cette matrice pour chaque sujet et une colonne pour chaque objet. Chaque cellule de cette matrice spécifie les accès autorisés par le sujet de la ligne sur l'objet de la colonne. Le contrôle d'accès consiste à garantir que seuls les modes d'accès spécifiés par la matrice sont réellement exécutés.

Le modèle matrice d'accès voit le système de sécurité en termes d'états. Un état est caractérisé par un ensemble de sujets, d'objets et d'autorisations. Des procédures d'analyse vérifient que les modifications de l'état du système (transitions) respectent bien l'intégrité de celui-ci.

	Fichier1	Fichier2	Fichier3	Fichier4
Personne1	*Own* *R* *W*		*Own* *R* *W*	
Personne2	*R*	*Own* *R* *W*	*W*	*R*
Personne3	*R* *W*	*R*		*Own* *R* *W*

TAB. 2.1 – Une exemple d'une matrice d'accès.

Le modèle Take-Grant qui utilise une structure de graphe pour représenter l'état du système, se concentre sur le problème de la propagation des privilèges. Des variations ont été proposées à ce modèle visant à affiner le contrôle de la propagation des privilèges. Le modèle Action-Entity considère un ensemble plus riche de privilèges administratifs et supporte des prédicats dans les autorisations (Figure 2.3).

Un **état d'autorisation** Φ est un triplet (S, O, A) où
- S est un ensemble de sujets (entités actives),
- O est un ensemble d'objets (entités passives ou actives), avec $S \subset O$,
- A est la matrice d'accès, $A[s, o]$ indique les modes d'accès que le sujet s peut effectuer sur l'objet o.

L'ensemble des **modes d'accès** dépend du type de l'objet à protéger et des fonctionnalités du systèmes. Les modes d'accès typiques pour les fichiers sont : **Lecture** (Read), **Ecriture** (Write), **Exécution** (Execute) et **Propriétaire** (Own). Le mode propriétaire (ownership) permet de contrôler qui peut changer les permissions d'accès (le Tableau 2.1).

Si une composante de la matrice $A[s, o]$ contient le mode propriétaire, alors s est considéré comme le propriétaire de o et par conséquent s possède les droits administratifs sur o.

L'état Φ du système peut être modifié par une séquence d'**opérations** qui modifient la matrice d'accès. Six opérations élémentaires ont été identifiées :

1. *ajouter un mode d'accès* r à la composante $A[s, o] \rightarrow grant$,
2. *retirer un mode d'accès* r de la composante $A[s, o] \rightarrow revoke$,
3. *créer un sujet* $s_k \rightarrow Ajout$ d'une ligne à la matrice A,
4. *supprimer un sujet* s_k,
5. *créer un objet* $o_k \rightarrow Ajout$ d'une colonne à la matrice A,
6. *supprimer un objet* o_k.

Le fait que le propriétaire puisse donner une autorisation sur un mode d'accès même si celui-ci ne le possède pas explicitement viole le principe d'atténuation des privilèges. Ce principe stipule qu'un sujet ne peut pas augmenter ses privilèges ou donner des privilèges qu'il ne possède pas.

Un exemple de ce type de matrice d'accès est montré dans le Tableau 2.1, où les droits d'accès sont lecture - R et écriture - W. Les sujets Personne1, Personne2, Personne3 possèdent des autorisations sur les fichiers - les objets. Il est spécifié par exemple dans cette matrice que Personne1 est le propriétaire du Fichier1 et du Fichier3. Il peut également lire et écrire sur le fichier Fichier1 ou le Fichier3, par contre il n'a pas d'accès aux fichiers Fichier2 et Fichier4.

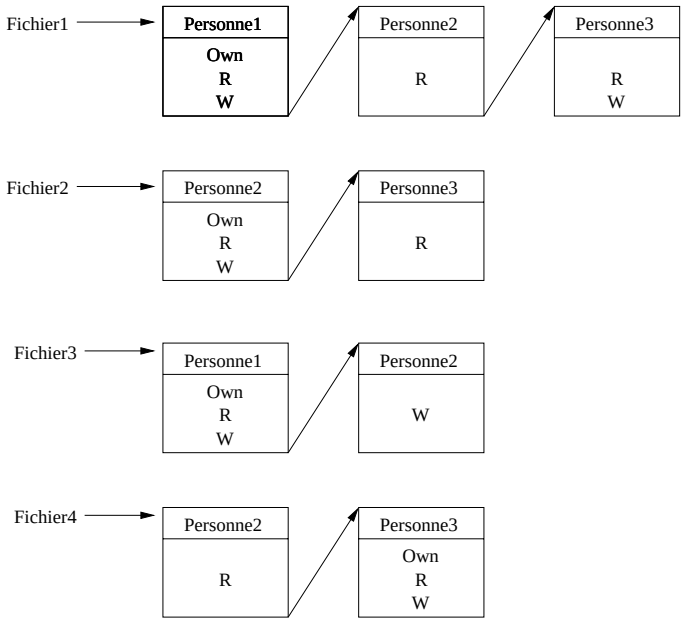


FIG. 2.4 – La liste de contrôle d'accès.

2.2.2 Implémentations possibles

Dans un système comportant de nombreux objets et sujets la matrice d'accès aurait des dimensions énormes et beaucoup de ses cellules seraient vides (i.e. matrice creuse). C'est pourquoi plusieurs implémentations moins coûteuses ont été proposées pour représenter cette matrice.

2.2.2.1 La liste de contrôle d'accès

Une approche très populaire pour implémenter une matrice d'accès sont les **Listes de Contrôle d'Accès - Access Control Lists** (ACLs). Chaque objet est associé à une ACL indiquant les autorisations sur cet objet pour chaque sujet du système. Cette approche correspond à une vision de la matrice par les colonnes (Figure 2.4) et privilégie un mode de consultation par les objets.

Par contre la recherche de tous les accès qu'un sujet possède, est plus difficile dans un système basé sur un ACL. On doit en effet examiner l'ACL de chaque objet pour les déterminer.

2.2.2.2 La liste de capacités

C'est l'approche duale de l'ACL. Chaque sujet est associé à une liste appelée une **liste de capacités**, déterminant les autorisations qu'il détient sur les objets du système. Cette approche correspond à une vision de la matrice d'accès par ligne (Figure 2.5) et privilégie donc un mode de consultation par les sujets.

Le même type de reproche que précédemment peut être fait si on désire connaître tous les sujets accédant à un objet particulier.

2.3 Le modèle à politique de contrôle d'accès mandataire (MAC)

La ***politique de contrôle d'accès mandataire*** gère l'accès aux informations par un individu en se basant sur la classification des sujets et des objets selon leur habilitation ou leur sensibilité [SS94, SS97, CFMS94]. Chaque sujet et chaque objet du système est associé à un *niveau de sécurité*. La politique mandataire peut être aussi vue comme une politique de contrôle du flux parce qu'elle

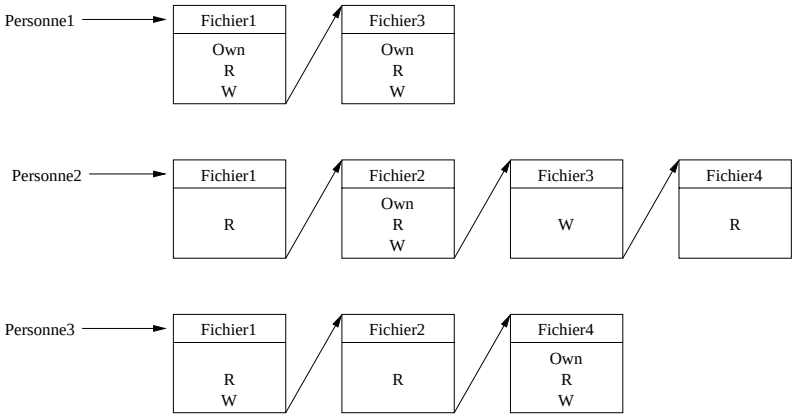


FIG. 2.5 – La liste de capacités.

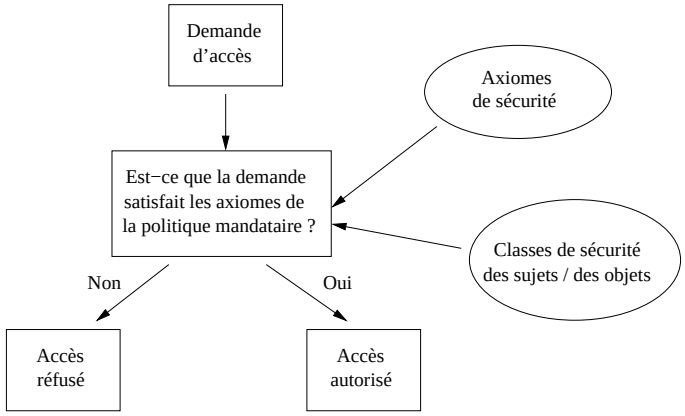


FIG. 2.6 – Le contrôle d'accès mandataire.

prévient des flux d'informations en empêchant par exemple la déclassification d'un objet (Figure 2.6).

Les objets sont des entités passives contenant l'information, telles que les fichiers de données, les enregistrements, les champs dans un enregistrement, etc. Les sujets sont des entités actives, telles que les utilisateurs qui accèdent aux objets.

Un niveau de sécurité est composé de deux éléments :

- un *niveau de classification* qui exprime un degré de sensibilité,
- une *catégorie* qui permet de fractionner l'espace ou le domaine d'application.

Le niveau de sécurité associé à l'objet reflète le niveau de sensibilité des informations contenues dans cet objet. Le niveau de sécurité associé à l'utilisateur, reflète la fiabilité de cet utilisateur - le niveau de confiance ou d'habilitation d'un utilisateur.

Les niveaux de classification sont ordonnés selon leur degré de sensibilité. Une hiérarchie consiste par exemple en :

- **Top Secret** (TS) - hautement secret,
- **Secret** (S) - secret,
- **Confidentiel** (C) - confidentiel,
- **Unclassified** (U) - ordinaire

avec TS $>$ S $>$ C $>$ U.

Les catégories ne sont pas ordonnées et indiquent les différents champs d'application du système

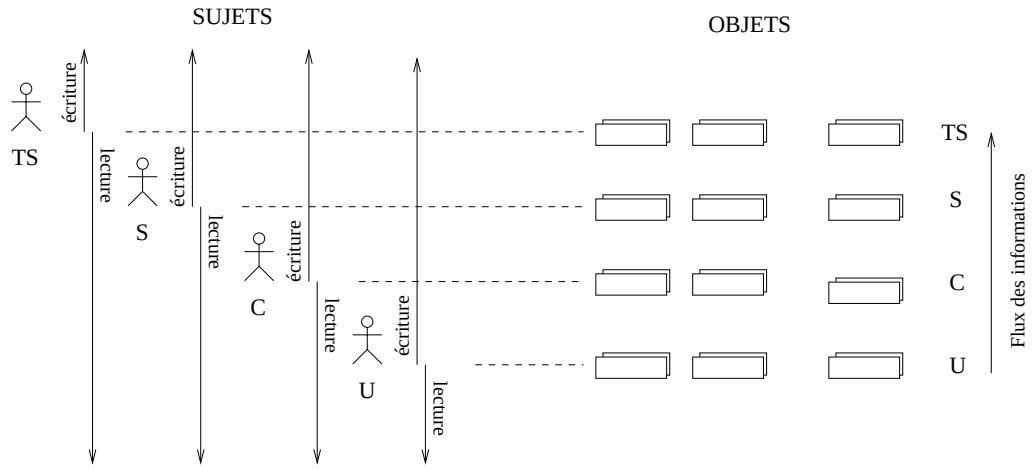


FIG. 2.7 – Le contrôle de flux des informations.

(entreprise, organisation). Dans un milieu industriel on peut définir les catégories suivantes :

Production - Personnel - Ingénieur - Administratif

L'ensemble des catégories associé à un utilisateur reflète des zones dans lesquelles cet utilisateur peut opérer.

Les niveaux de sécurité peuvent être comparées deux à deux dans certain cas. Une relation d'ordre partiel appelée "*domine*" est définie sur l'ensemble des niveaux de sécurité. On note $\geq$ la relation *domine* et $>$ la relation *domine strictement* : le niveau de sécurité $L_1 = (S_1,\ C_1)$ domine le niveau de sécurité $L_2 = (S_2,\ C_2)$, si et seulement si : $S_1 \geq S_2\ et\ C_1 \supseteq C_2$.

L'accès à un objet par un sujet est assuré si et seulement si une relation, dépendant du type d'accès, est remplie entre les niveaux de sécurité associés à ceux-ci. En particularité, il y a deux principes ou contraintes qu'on doit vérifier [SS94] :

1. **Lecture en bas** (Read down) - le niveau de sécurité du sujet doit dominer le niveau de sécurité de l'objet à lire. Pour lire une information la classification du sujet doit dominer celle de l'objet.
2. **Ecriture en haut** (Write up) - le niveau de sécurité du sujet doit être dominé par le niveau de sécurité de l'objet à écriture. Pour écrire dans un fichier, la classification de l'objet doit dominer celle du sujet.

La satisfaction de ces principes permet de se protéger contre la propagation des informations d'un objet d'un niveau supérieur vers un objet de niveau inférieur. Elle permet ainsi de garantir la confidentialité des informations (Figure 2.7). Le flux d'informations est donc ascendant.

Il y a malgré tout quelques inconvénients à ce principe. Un utilisateur de niveau de confiance S peut faire une écriture sur un objet ayant un niveau de sensibilité TS même si la lecture est interdite par le premier principe. Ce qui revient à dire qu'un utilisateur est capable de détruire des données de niveau supérieur. Le même utilisateur (avec le niveau de confiance S) ne peut pas faire une écriture sur un objet de niveau de sensibilité C ou U. Ce qui veut dire que cet utilisateur ne pourra jamais envoyer un message non stratégique aux utilisateurs de C ou U (Figure 2.8).

Le contrôle d'accès mandataire peut être utilisé pour la protection de l'intégrité des information. Les niveaux de l'intégrité peuvent être les suivants : **Décisif** (Crucial (C)), **Important** (Important (I)) et **Inconnu** (Unknown (U)). Le niveau de l'intégrité associé à un objet reflète le degré de confiance des informations déposées dans cet objet.

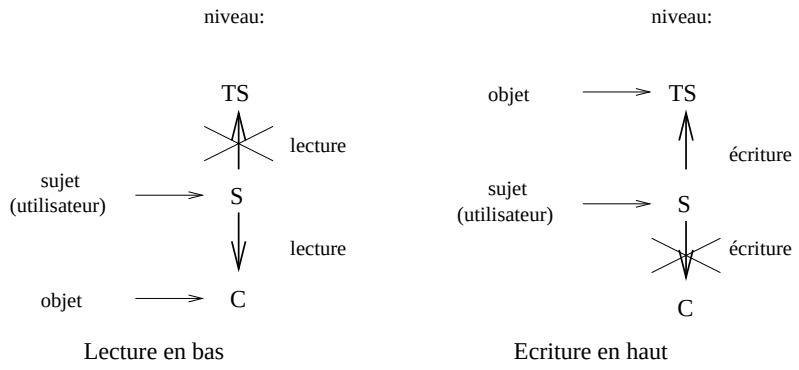


FIG. 2.8 – Les deux principes - lecture en bas et écriture en haut.

Les modèles à politique mandataire suivants sont parmi les plus connus : le modèle de Bell and LaPadula [BP75], le modèle de Biba [Bib77] ou le modèle de Jajodia and Sandhu [SJ93]. La prochaine section décrit le modèle de Bell-LaPadula.

2.3.1 Le modèle de Bell-LaPadula

Ce modèle multi-niveaux fut initialement proposé par D. E. Bell et L. J. La Padula en 1973.

Le modèle est basé sur la classification des éléments du système. La classification est exprimée par des niveaux de sécurité. Chaque niveau de sécurité est défini par deux éléments : sa classification et sa catégorie. Un niveau de sécurité $L_1 = (S_1, C_1)$ est supérieur ou égal au niveau $L_2 = (S_2, C_2)$ si les relations $S_1 \geq S_2$ et $C_1 \supseteq C_2$ sont respectées. Les relations peuvent être strictes.

Un niveau de sécurité appelé habilitions ("clearance") est assigné à chaque utilisateur. Le modèle contient les types de mode d'accès suivants :

- *read-only* - pour lire des informations contenues dans un objet,
- *append* - pour ajouter des informations dans un objet sans voir les informations qu'il contient,
- *execute* - pour exécuter un objet (un programme),
- *read-write* - pour écrire dans un objet, ce mode permet aussi à un sujet de lire le contenu d'un objet.

Chaque droit d'accès sur un objet est transférable ("grant") d'un sujet à un autre. Le créateur d'un objet est appelé le propriétaire de cet objet. Seul le propriétaire d'un objet peut transférer un droit, sauf le mode propriétaire qui est incessible. Ce modèle supporte une gestion décentralisée des privilèges.

L'état du système peut être changé par l'exécution de huit opérations différentes :

- *get access (demander un accès)* - sur un objet pour un mode d'accès,
- *release access (libérer un accès)* - opération inverse de la précédente,
- *give access (donner un accès)* - pour donner un accès ("grant") à un sujet sur un objet,
- *rescind access (aborder un accès)* - opération inverse de la précédente,
- *create object (créer un objet)* - pour créer un nouvel objet dans la hiérarchie des objets ; l'objet est soit actif si des droits lui sont attachés à des sujets, soit inactif si aucun droit ne lui est attaché,
- *delete object (détruire un objet)* - opération inverse de la précédente, l'objet est retiré de la hiérarchie,
- *change subject security level (changer le niveau de sécurité d'un sujet)* - l'exécution de cette opération modifie le niveau de sécurité attaché au sujet,
- *change object security level (changer le niveau de sécurité d'un objet)* - l'exécution de cette opération modifie le niveau de sécurité attaché à un objet inactif ; le niveau niveau de l'objet doit être dominé par le niveau de sécurité du sujet qui demande l'opération.

FIG. 2.9 – La notion de rôle.

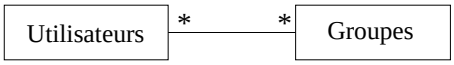


FIG. 2.10 – La notion de groupe.

Le modèle définit un certain nombre d'axiomes de sécurité qui doivent être vérifiés pour tout état du système. Chaque opération est contrôlée par un programme de surveillance. Le modèle développe la notion de sujet de confiance ("trusted subject") : sujet ne pouvant pas compromettre la sécurité du système.

2.4 Le modèle de contrôle d'accès basé sur les rôles (RBAC)

Un ***modèle RBAC*** est une alternative aux modèles de contrôle d'accès discrétionnaires et mandataires. Ce modèle est apparu initialement avec les systèmes multi-utilisateurs et multi-applications des années soixante dix [Cod70, Cod76, FK92, Gui95, FCK95, FBK99, SCFY96].

Le concept de base de ces modèles est d'associer des permissions à des rôles, puis d'assigner les utilisateurs aux rôles appropriés [San93, SS94, San96, SM98a] - Figure 2.9.

Ces politiques demandent l'identification des différents rôles qu'auront les acteurs du système d'information. Un *rôle* représente une compétence pour réaliser une tâche particulière dans l'entreprise. Il peut être vu comme la possibilité qui est donnée à un utilisateur possédant ce rôle de pouvoir exécuter les actions et les responsabilités associées à une activité particulière dans une organisation.

Un rôle permet donc à un ensemble d'utilisateurs de partager un ensemble de permissions. Un rôle est donc un concept organisationnel puissant pour mettre en place une politique de sécurité. Il offre beaucoup plus de flexibilité au niveau de ses associations que les modèles précédents.

Un *utilisateur* jouant un rôle peut exécuter tous les accès pour lesquels ce rôle a été conçu. Un utilisateur peut prendre différents rôles à des moments différents et le même rôle peut être pris par plusieurs utilisateurs, parfois en même temps.

Les *permissions* sont associées aux rôles et expriment les autorisations sur le système.

Globalement le travail de l'administrateur s'en trouve grandement simplifié si on le compare avec une politique de type discrétionnaire où les permissions de chaque utilisateur sont à gérer individuellement. C'est un concept beaucoup plus puissant que le concept de groupe que l'on trouve dans les systèmes discrétionnaires et qui ne représente qu'une liste d'utilisateurs (Figure 2.10). Une permission peut ensuite être affectée à un groupe d'utilisateurs.

L'approche basée sur les rôles possède beaucoup de qualité [SCFY96, San98, Mof98] :

1. *Gestion d'autorisation* - la politique RBAC apporte une indépendance logique de la spécification des autorisations, par rapport aux utilisateurs. Cet avantage est la conséquence du partage de la spécification des autorisations en deux parties. Les rôles sont d'abord crées pour les différentes fonctions ou tâches dans une organisation et les utilisateurs sont ensuite attachés aux rôles en se basant sur leurs responsabilités et leurs qualifications. Ce partage simplifie la gestion de la sécurité du système. Le rôle d'un utilisateur peut lui être retiré et de nouveaux rôles peuvent lui être attribués par rapport à ses nouvelles responsabilités dans une organisation. L'avantage de cette attribution est la facilité et le gain de temps pour la réalisation de cette tâche.

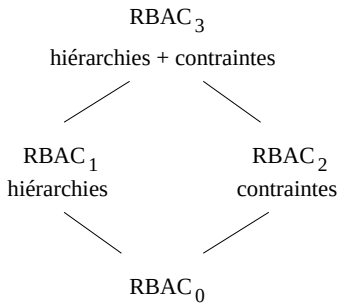


FIG. 2.11 – L'ensemble des modèles RBAC.

2. *Hiérarchie de rôles* - il existe très souvent une hiérarchie naturelle de rôles dans une organisation. Des relations d'héritage peuvent se mettre en place pour permettre à un rôle de niveau supérieur d'hériter des autorisations de ses rôles subordonnés.
3. *Privilèges minimum* ou *moindre privilège* - les rôles permettent au utilisateur de se connecter au système d'information avec les privilèges strictement nécessaires à une tâche particulière. Pour cela, un utilisateur active le rôle qui lui a été conçu pour remplir cette tâche.
4. *Séparation des responsabilités* - elle est utilisée pour empêcher les abus de pouvoir par un utilisateur ayant trop de privilèges. L'idée est de partager une transaction critique en deux actions différentes et de s'assurer que leurs exécutons ne puissent être faites par la même personne. La séparation des responsabilités peut être statique (des rôles mutuellement incompatibles sont définis par une contrainte empêchant l'affectation de même utilisateur à ces rôles) ou dynamique (force un contrôle dans le temps, au moment de l'activation du rôle).
5. *Classe d'objets* - un modèle RBAC permet aussi de classifier les utilisateurs selon leurs domaines d'activités et une classification pour les objets. Les objets peuvent être classifiés par rapport à leurs types ou leurs champs d'application.

2.4.1 Les modèles RBAC

Les modèles de R. S. Sandhu sont parmi les propositions les plus connus [SCFY96]. Il a défini une famille de quatre modèles conceptuels RBAC (Figure 2.11) :

1. $RBAC_0$ - il spécifie les exigences minimales pour supporter complètement un modèle RBAC de base,
2. $RBAC_1$ - c'est un modèle $RBAC_0$ auquel on a ajouté le concept d'héritage de rôles qui permet à un rôle d'hériter des permissions d'autres rôles,
3. $RBAC_2$ - c'est un modèle $RBAC_0$ auquel on a ajouté des contraintes qui imposent des restrictions sur l'affectation des différents composants d'un modèle RBAC,
4. $RBAC_3$ - un modèle consolidé, qui réunit le modèle $RBAC_1$ et $RBAC_2$ et par transitivité le modèle $RBAC_0$.

Une mise en oeuvre de cette famille de modèles repose (Figure 2.12) :

1. sur trois collections d'unités :
   (a) **utilisateurs (U)** - un utilisateur est un être humain ou un agent autonome,
   (b) **rôles (R)** - un rôle est une fonction ou un titre de travail dans une organisation ayant une sémantique associée en rapport avec l'autorité et la responsabilité conférée à un membre de ce rôle,
   (c) **permissions (P)** - une permission est une autorisation d'un mode particulier d'accès sur un ou plusieurs objets du système. Les permissions sont toujours positives. Les

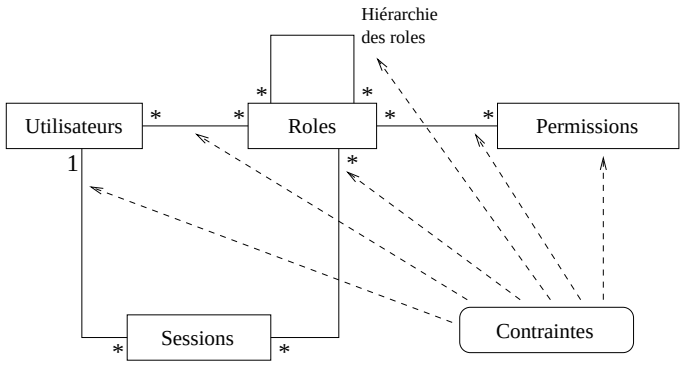


FIG. 2.12 – Un modèle RBAC (en utilisant la notation UML).

interdictions seront modélisées dans le modèle $RBAC_2$ par des contraintes et non par des autorisations négatives,

2. sur des relations
   On désignera par **U-R** les relations d'association entre les utilisateurs et les rôles et par **R-P** les relations entre les rôles et les permissions. Ces types de relations sont toutes les deux des relations "many-to-many". Un utilisateur peut être membre de plusieurs rôles et plusieurs utilisateurs peut être affecté à un rôle. De la même façon, un rôle peut être associé à plusieurs permissions et la même permission peut être attachée à plusieurs rôles. Il existe aussi une relation entre les rôles (notée **R-R**) qui permet à un rôle d'hériter des permissions d'un autre rôle. Un rôle hérite des permissions attachées aux différents rôles fils. L'héritage dans une hiérarchie de rôles est transitif et l'héritage multiple est également possible [San96, San98].
3. sur des sessions
   Chaque **session (S)** représente le profil dynamique d'un utilisateur. Un utilisateur établit une session durant laquelle il active un sous-ensemble de rôles dont il est un membre directement ou indirectement par rapport à une hiérarchie de rôles. L'association entre session et rôle indique que des rôles multiples peuvent être activités dans le même temps, c'est-à-dire dans la même session. Les permissions possibles pour cet utilisateur est l'union des permissions de tous rôles activés dans cette session. Chaque session est associée à un seul utilisateur comme l'indique l'association 1..* de la Figure 2.12.
   Un utilisateur peut ouvrir plusieurs sessions en même temps avec une combinaison différente de rôles activés pour chaque session. Les rôles peuvent être aussi activés ou désactivés dynamiquement pendant la durée de vie d'une session [SCFY94, SS97, JSpSB98, SFK00].

2.4.2 Contraintes

Les interdictions (autorisations négatives) sont modélisées par des *contraintes*. Les contraintes peuvent se rapporter à tous les composants du modèle. C'est un mécanisme puissant qui garantit une structure organisationnelle de haut niveau [San90, SCFY96, CS95].

Les contraintes peuvent être appliquées sur toutes les relations du modèle RBAC. Les contraintes sont des prédicats qui appliqués à ces relations et/ou composants, retournent la valeur "*satisfaite*" (accepté) ou la valeur "*non satisfaite*" (non accepté).

Les contraintes qui sont mentionnées le plus souvent dans le modèle RBAC sont :
– Les *contraintes d'une séparation de responsabilité* - une contrainte se base sur un concept des rôles qui s'excluent mutuellement, c'est-à-dire une contrainte entre plusieurs rôles interdit à un utilisateur d'appartenir simultanément à plus d'un de ces rôles. Par exemple, on peut rendre exclusifs les deux rôles *ResponsableDeVente* et *ResponsableDuPaiement* empêchant

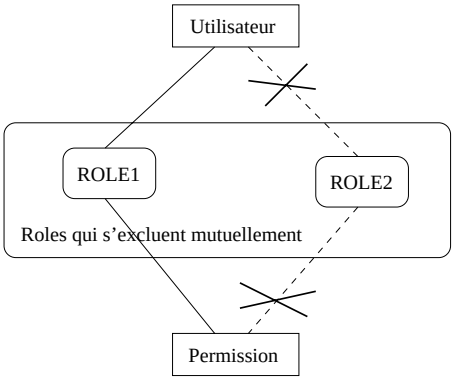


FIG. 2.13 – Les rôles mutuellement exclusifs.

un même utilisateur d'être membre de ces deux rôles. Cela permet une séparation de responsabilités.
Une contrainte d'exclusion mutuelle peuvent être appliquée aussi sur la relation R-P (une attribution d'une permission). La permission ne pourra pas être attribuée à deux rôles appartenant à un même ensemble de rôles mutuellement exclusifs. Par exemple la permission de recevoir des chèques ne peut pas être attachée à la fois aux rôles *ResponsableDeVente* et *ResponsableDuPaiement* (Figure 2.13).

- Les *contraintes de cardinalité* - cette contrainte détermine par exemple le nombre d'utilisateurs maximum qui peuvent être attachés au rôle. Le nombre de rôles auxquels un utilisateur individuel peut aussi être limité. Le nombre de rôles auxquels une permission peut être attachée, peut être également limité.
- Les *contraintes de pré-requis* - ce concept est basé sur la possibilité qu'un utilisateur puisse être attaché au rôle r_2 si l'utilisateur est déjà membre du rôle r_1. Par exemple seuls les utilisateurs qui sont déjà membres d'un rôle *Employé* peuvent être attachés au rôle *Ingénieur*. La même situation existe pour l'affectation des permissions à un même rôle.
- Les *contraintes de sessions* - le système accepte qu'un utilisateur puisse être attaché à deux rôles mais il ne pourra pas les activer simultanément dans la même session. Une autre contrainte peut limiter le nombre de sessions qu'un utilisateur peut activer dans le même temps.
- Les *contraintes sur une hiérarchie de rôles* - par exemple une permission attachée au rôle fils ne peut pas être attachée au rôle parent.

2.4.3 La définition formelle du modèle RBAC

La définition suivante formalise ce modèle RBAC [SCFY96] :
Un modèle RBAC possède les composantes suivantes :

- U, R, P et S, où U - un ensemble d'utilisateurs, R - un ensemble de rôles, P - un ensemble de permissions et S - un ensemble de sessions,
- $R - P \subseteq R \times P$ - une relation d'attribution d'une permission à un rôle qui est de type "plusieurs-vers-plusieurs",
- $U - R \subseteq U \times R$ - une relation d'attribution d'un utilisateur à un rôle qui est de type "plusieurs-vers-plusieurs",
- $R - R \subseteq R \times R$ - une hiérarchie de rôles qui définit un ordre partiel (une relation réflexive - un rôle hérite de ses propres permissions, transitive et antisymétrique) à l'aide d'une relation *domine* notée $\geq$,
- $user : S \rightarrow U$ - une fonction qui retourne l'utilisateur associé à une session s_i, constante pour le cycle de vie de la session,

- $roles : S \rightarrow 2^{R}$ - une fonction qui retourne un ensemble de rôles activés dans une session s_i : $roles(s_i) \subseteq \{r \mid (\exists r' \geq r)[(user(s_i), r') \in U - R]\}$ qui peut changer pendant le cycle de vie de la session s_i. L'utilisateur possède les permissions $\bigcup_{r \in roles(s_i)} \{p \mid (p, r) \in U - R\}$,
- un ensemble de contraintes qui détermine si les valeurs des différents composants d'un modèle RBAC seront acceptées.

2.4.4 Le modèle d'administration - le modèle ARBAC

Pour gérer l'administration des rôles, deux types d'éléments ont été ajoutés par les auteurs du modèle RBAC : les rôles administratifs AR et les permissions administratives AP.

Les permissions administratives AP permettent de changer les différents composants contenus dans les modèles $RBAC_0$, $RBAC_1$, $RBAC_2$ ou $RBAC_3$. Les permissions administratives sont disjointes des permissions régulières P, c'est-à-dire $AP \cap P = \emptyset$. Les rôles administratifs sont également disjoints des rôles réguliers R, c'est-à-dire $AR \cap R = \emptyset$ [Mun00].

Ce modèle précise que les permissions peuvent être seulement attachées aux rôles et les permissions administratives peuvent être seulement attachées aux rôles administratifs.

Les autorités d'un administrateur de sécurité lui confèrent le droit de modifier les relations présentées précédemment. Les permissions qui autorisent ces opérations administratives doivent être définies dans un modèle de gestion.

Ce modèle d'administration est appelé le modèle ARBAC (Administrative RBAC) et il contient trois composant [BS97, SB97, SM98b, SaQM99, SM99] :

1. **user-role assignment - le modèle URA** - il est responsable de l'administration des relations U-R, c'est-à-dire de l'attachement des utilisateurs aux rôles,
2. **permission-role assignment - le modèle PRA** - il est responsable de l'administration des relations R-P, c'est-à-dire de l'attachement des permissions aux rôles,
3. **role-role assignment - le modèle RRA** - il définit trois types de rôles dans le modèle d'administration RBAC et il est responsable de l'héritage de rôles.

Le modèle RBAC donne un support pour implémenter des politiques de sécurité différentes mais ne dit pas comment ces principes sont à adapter dans la pratique. Une propriété importante de RBAC est sa neutralité d'un point de vue politique. Il est tout à fait possible avec RBAC de décrire des modèles de type MAC ou DAC [SM98a].

2.5 L'ingénierie des rôles d'un système d'information

Définir un modèle RBAC, revient à déterminer les rôles de chacun au sein d'une organisation. Une bonne organisation de l'entreprise est donc une condition sine qua non pour la réussite du modèle RBAC.

2.5.1 Les méthodes de conception des systèmes d'information et de sécurité

Nous avons présenté, dans le Chapitre 1, deux méthodes de conception et de développement des systèmes d'information, Merise et Ossad. Ces méthodes possèdent des éléments qui permettent de modéliser l'organisation de l'entreprise, en particulier de préciser l'accès aux données (Merise) ou d'introduire la notion de rôle (Ossad).

En Merise, dès le niveau organisationnel la détermination des droits d'accès doit être traitée. Pour chaque donnée ou ensemble de données, les droits d'accès sont définis par type d'acteur et/ou par type de site dans le cas d'un système d'information multi-sites. Les modes d'accès se déclinent en droits de consultation de données et droits de mise à jour de données.

Dans le cas d'un système d'information organisé en plusieurs sites, le niveau organisationnel permet de décrire la visibilité propre à chaque site en identifiant les données concernées par celui-ci.

Les modes d'accès en consultation. Un accès aux données en consultation est autorisé (par défaut) sauf dans le cas spécifique où il s'agit de données sensibles (i.e. confidentielles). Les données peuvent avoir un niveau de consultation limité pour des raisons stratégiques propres à chaque entreprise, par exemple les informations sur les salaires ou les comptes financiers. L'étude des droits d'accès en consultation doit se conclure par un tableau indiquant les données accessibles par type d'acteur.

Les modes d'accès en mise à jour. Les droits d'accès en mise à jour représentent un enjeu fondamental dans la construction d'un système d'information. Il faut définir les droits de mise à jour par ensembles homogènes et cohérents de données et par type d'acteur responsable.

Merise est donc une bonne approche globale du système d'information avec une description riche au niveau conceptuel fondée sur les invariants du système d'information, mais elle ne donne pas de recommandation particulière pour la mise en place de politiques de sécurité comme RBAC.

Aucune méthode ne se préoccupe vraiment de la spécification d'un modèle de contrôle d'accès. Cependant, Merise permet de spécifier les modes d'accès aux données dans le niveau organisationnel. Quant à la méthode Ossad, la notion de rôle (ensemble de responsabilités), d'équipe (ensemble de rôles), la matrice d'activités-rôles et le graphe de rôles sont des éléments de modélisation sur lesquels une politique de contrôle d'accès pourrait s'appuyer.
Cependant, elles restent insuffisantes pour la spécification de politique de sécurité basée sur les rôles.

Nous avons aussi présenté, dans le Chapitre 1 des méthodes dédiées pour la sécurité des systèmes d'information, Marion et Méhari. Ces méthodes sont très globales et elles permettent d'une certaine manière la modélisation de la sécurité logique, c'est-à-dire du contrôle d'accès. Cependant, elles ne déterminent pas directement des éléments pour mettre en place un processus d'ingénierie des rôles.

2.5.2 Les méthodes basées sur le modèle RBAC

Le modèle RBAC est principalement centré sur la notion de rôle plutôt que sur la notion d'utilisateur. On rappelle que le modèle de base demande d'identifier les rôles, de leur attacher des permissions puis de leur affecter finalement des utilisateurs.
Beaucoup de travaux ont défini les aspects formels et structurels des rôles et leurs droits [FK92, FCK95, Gui95, SCFY96, SFK00]. Cependant, il y a peu de travaux qui déterminent comment construire et gérer les rôles nécessaires à un système d'information [FH97, TOB98, JSpSB98, ES99, FBK99].
Comment mettre en oeuvre sur le terrain les rôles d'un modèle RBAC ? Quelle méthodologie choisir pour les déterminer efficacement et simplement ? Bref, qu'en est-il de l'ingénierie des rôles ?

2.5.2.1 Le modèle NIST-RBAC

Sandhu et all. propose dans [SFK00] un standard pour le modèle RBAC - le modèle NIST-RBAC. Ce modèle organisé en couches possède quatre niveaux ayant des fonctionnalités bien définies. Chaque niveau possède les propriétés des niveaux inférieurs. Les niveaux du modèle NIST-RBAC sont définis comme suit (Figure 2.14) :

1. **Flat RBAC** - contient les concepts RBAC les plus importants et les plus essentiels : les utilisateurs sont attachés aux rôles, les permissions sont attachées aux rôles et les utilisateurs possèdent les permissions parce qu'ils sont membres de rôles. Le modèle exige que les relations U-R et P-R sont de type "plusieurs-à-plusieurs" et le même utilisateur peut être attaché à

Flat RBAC	relation U−R et relation P−R de type "plusieurs−à−plusieurs utilisateurs possèdent les permissions par les roles utilisateurs peuvent exécuter les permissions de differents roles simultanement
Hierarchical RBAC	Flat RBAC + hierarchie de roles
Constrained RBAC	RBAC d'hierarchie + SOD
Symetric RBAC	RBAC de contraintes + support pour la relation P−R

FIG. 2.14 – Les quatre niveaux du modèle NIST-RBAC.

plusieurs rôles. Les utilisateurs peuvent prendre différents rôles en même temps. Ces exigences de Flat RBAC sont obligatoires pour tous les niveaux suivants.

2. **Hierarchical RBAC** - ajoute la notion d'une hiérarchie de rôles. La hiérarchie de rôles peut être de type de la hiérarchie d'héritage (un utilisateur qui possède un rôle père, possède automatiquement tous les rôles fils) et/ou de type de la hiérarchie d'activité (un utilisateur qui active un rôle père, n'a pas automatiquement les rôles fils).
3. **Constrained RBAC** - ajoute la séparation des responsabilités (SOD).
4. **Symetric RBAC** - ajoute une spécification précise de la relation permission - rôle.

Cet article propose une standardisation du modèle RBAC, mais ne traite pas de l'ingénierie des rôles au niveau du modèle RBAC.

2.5.2.2 L'environnement RBAC-FNE

Thomsen et all. [TOB98] proposent un modèle RBAC nommé ***Role Based Access Control Framework for Network Enterprises (RBAC-FNE)***. Ce modèle est représenté par sept niveaux d'abstraction qui sont sous la responsabilité des concepteurs d'application et des administrateurs locaux du système. Ces sept niveaux sont divisés en deux catégories (Figure 2.15) :

1. le concepteur de l'application est responsable de la création de quatre premiers niveaux ; c'est en effet lui qui connaît le mieux l'application, donc les contraintes associées à celle-ci,
2. l'administrateur de sécurité du système est responsable de la gestion de trois niveaux supérieurs ; c'est lui qui connaît le mieux le schéma directeur de sécurité de l'entreprise, donc les contraintes de celle-ci.

Les quatre premiers niveaux sont les blocs de base du modèle. Le composant de base est l'*objet*. Un objet possède un nom et un ensemble de méthodes publiques qui peuvent être utilisées pour accéder au objet. Ces méthodes peuvent être considérées comme les modes nécessaires pour réaliser des actions sur un objet. Les attributs de l'objet peuvent être utilisés pour établir des contraintes.

Habituellement un objet possède plusieurs méthodes qui seront souvent accèdées ensemble. Il est plus intéressant de grouper ces méthodes dans un ensemble qui sera ensuite attaché à un rôle plutôt que de les attacher séparément. Ces *poignées d'objets* possèdent une description textuelle qui définit le type d'accès représenté par cet ensemble de méthodes et comment l'utiliser.

La politique de sécurité doit aussi exprimer quels individus possèdent un accès aux objets précis. Donc, il faut spécifier un ensemble de conditions au moment où une poignée d'objet est

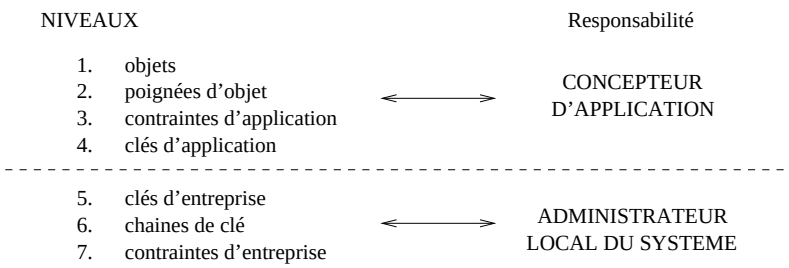


FIG. 2.15 – Les niveaux de l'environnement RBAC-FNE.

attachée à un rôle. Ces conditions doivent être satisfaites avant d'accéder aux méthodes d'une poignée donnée et elles sont définies au niveau des *contraintes d'application*.

Le dernier niveau du concepteur est le niveau des *clés d'application*. Un utilisateur qui a obtenu une clé peut accéder aux ressources représentées par cette clé. Une clé d'application est un ensemble de méthodes et de contraintes d'application sur ces méthodes. Les clés d'application peuvent être considérées comme les rôles d'une application. Les clés peuvent être définies dans une hiérarchie semblable à une hiérarchie de rôles.

Les trois niveaux suivants - les niveaux d'entreprise - sont gérés par l'administrateur de sécurité. Le premier niveau est utilisé pour créer les *clés d'entreprise*. Chaque clé d'application qui représente un rôle non-abstrait est attachée à une clé d'entreprise. Les utilisateurs sont soit attachés aux clés d'entreprise à ce niveau ou soit attachés à une chaîne de clés définie au niveau supérieur. Une clé d'entreprise permet à l'utilisateur d'accéder aux méthodes associées si et seulement si les contraintes d'application sont satisfaites.

L'administrateur de sécurité groupe les clés d'entreprise en ensembles cohérents appelés *chaînes de clés*. Une chaîne de clés peut être attachée à un utilisateur. Une chaîne de clés peut contenir d'autres chaînes de clés. L'administrateur peut contrôler la politique de sécurité par la définition de la structure des chaînes de clés et par l'affectation des utilisateurs à ces chaînes.

Le dernier niveau est utilisé pour définir les *contraintes d'entreprise*. Elles sont liées avec les chaînes de clés pour prévenir des accès non-autorisés des utilisateurs aux données des applications.

Ce modèle permet le partage des éléments d'un système d'information par rapport aux besoins de chacun et aux stratégies de sécurité qui ont été définies. Il est basé sur une approche du modèle RBAC classique.

Le concept de clé d'application est proche du concept de rôle et donc un processus d'ingénierie des rôles peut être réalisé par l'étape de définition des clés d'application, des droits de ces clés et de leur affectation aux utilisateurs.

Cependant, cet environnement ne contient pas tous les concepts du modèle RBAC, comme la séparation des responsabilités ou le moindre de privilège, qui doivent être encore introduits et implémentés.

2.5.2.3 Une représentation UML de l'environnement RBAC-FNE

P. Epstein et R. Sandhu décrivent dans [ES99] une représentation de l'environnement présenté ci-dessus en utilisant le langage UML [Gro97, Gro00a]. Chaque niveau de l'environnement RBAC-FNE est défini par les éléments d'UML suivants :

1. *niveau objet* - les objets possèdent des méthodes publiques pour accéder au objet ; les attributs et les méthodes privées sont cachées ; une *classe* du langage UML contient le nom de l'objet et les opérations qui peuvent être privées ou publiques,
2. *niveau poignée d'objet* - les méthodes liées dans une poignée possèdent les caractéristiques d'une *interface*. On peut établir la relation entre une classe et une interface pour créer une poignée d'objet,

	Roles	Permissions	Hierarchie	Contrainte	Utilisateur
niveau 1		X			
niveau 2		X			
niveau 3		X		X	
niveau 4	X		X		
niveau 5	X				X
niveau 6	X			X	X
niveau 7	X			X	

TAB. 2.2 – Les éléments du modèle RBAC par rapport à l'environnement RBAC-FNE.

3. *niveau contrainte d'application* - une contrainte d'application peut être présentée comme une *pré-condition* avant l'exécution d'une opération,
4. *niveau clé d'application* - ce niveau permet de représenter une hiérarchie de rôles, donc une hiérarchie de poignées d'objet. Les clés d'application sont présentées comme des *paquetages* du langage UML dans lesquels les interfaces sont liées par des relations de généralisation,
5. *niveau clé d'entreprise* - on définit à ce niveau les clés d'entreprise ; elles sont représentées comme des *paquetages* du langages UML et elles sont attachées aux acteurs (utilisateurs),
6. *niveau chaîne de clés* - une chaîne de clés contient un ensemble de clés d'entreprise ; les clés sont liées dans un paquetage pour créer une chaîne de clés et cette chaîne est attachée à un acteur (utilisateur),
7. *niveau contrainte d'entreprise* - une contrainte est définie dans une note qui est attachée à une chaîne de clés.

Les éléments provenant du modèle RBAC sont rappelés dans le Tableau 2.2 par rapport aux différents niveaux de l'environnement RBAC-FNE.

Cette proposition est seulement une spécification UML du modèle RBAC-FNE précédent. Elle n'apporte rien de plus au niveau du processus d'ingénierie des rôles.

2.5.2.4 Une méthode basée sur les cas d'utilisation

Une méthode pour déterminer des droits associés à un rôle est présentée dans [FH97]. Elle se base sur les cas d'utilisation et les séquences de cas d'utilisation. Cette approche garantit que chaque rôle reçoit seulement les droits nécessaires pour réaliser telle fonction dans un système. Un administrateur de sécurité définit des droits d'autorisation en se basant sur tous les cas d'utilisation du système.

Une règle d'autorisation est présentée sous forme de (S, O, T, P), où S représente le sujet (utilisateur ou rôle), O - l'objet accédé, T - le type d'accès permis et P - un prédicat optionnel qui définit des contraintes d'accès.

Un cas d'utilisation permet de trouver les classes et les opérations de ces classes qui seront manipulées par un acteur (rôle). Les autorisations doivent être attachées aux cas d'utilisation pour accéder aux opérations spécifiques d'un objet. Les cas d'utilisation étudient toutes les fonctions possibles d'utilisation du système. On peut donc déduire tous les droits demandés par un rôle en considérant les méthodes qui doivent être exécutées par l'acteur correspondant.

Avec un diagramme de scénario associé pour un cas d'utilisation (Figure 2.16) une définition formelle d'un droit de rôle est donnée comme suit :

un droit d'autorisation R est présenté par $R\,(A_i,\ M_j,\ O_k)$ où A_i est un acteur, M_j est une méthode et O_k est un objet. Un droit d'accéder à un objet O_k pour un acteur précis A_i en utilisant une méthode M_j peut être décrit comme suit :

$$R_{j,k}\,(A_i) = (M_j,\ O_k)$$

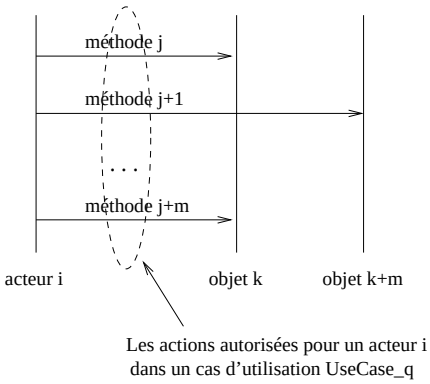


FIG. 2.16 – Un diagramme de scénario pour un UseCase_q [FH97].

Il existe pour chaque acteur un ensemble de droits venant d'un cas d'utilisation U_q. Cet ensemble de droits pour m méthodes dans un cas d'utilisation U_q est :

$$R_{U_q}(A_i) = \bigcup_{j_q, k_q = 1...m} \{R_{j_q,k_q}(A_i)\}$$

L'union des ensembles de droits de tous les cas d'utilisation pour un acteur particulier crée un ensemble complet de droits d'autorisation pour cet acteur pour n cas d'utilisation :

$$R(A_i) = \bigcup_{q = 1...n} \{R_{U_q}(A_i)\}$$

Cet ensemble de droits donne les permissions sur lesquelles un rôle pour un acteur A_i pourra être créé. L'union de ces ensembles de droits associés aux acteurs définit l'ensemble complet de droits d'autorisation pour le système étudié.

L'administrateur de sécurité doit donc analyser tous les cas d'utilisation et leurs pré-conditions pour décider quelles autorisations sont nécessaires pour chaque rôle. L'ajout ou la suppression d'une autorisation est seulement nécessaire si un cas d'utilisation est ajouté ou supprimé ou si celui-ci a fait l'objet d'une modification.

Cette approche propose le concept de cas d'utilisation pour déterminer les autorisations associées aux rôles définis dans une entreprise. Elle se base sur les besoins des personnes d'une entreprise exprimés au moyen des cas d'utilisation du système.

Le processus de définition des autorisations est réalisé par un parcours de tous les éléments (méthodes et objets) figurant dans les cas d'utilisation. Elle garantit la règle de moindre privilège, les utilisateurs étant attachés uniquement aux rôles correspondant à leurs fonctions dans l'organisation.

2.6 Conclusion

Le concept de protection des données comprend la protection contre les modifications et les divulgations non autorisées ou impropres. L'authentification, le contrôle d'accès, l'audit constituent ensemble la base pour construire des systèmes qui peuvent stocker et élaborer des informations avec un niveau de confidentialité et d'intégrité donné.

On a présenté dans ce chapitre plusieurs modèles de contrôle d'accès. Une politique de protection discrétionnaire gère et limite les accès aux données en utilisant l'identification d'un utilisateur et en déterminant les accès possibles et autorisés de cet utilisateur. Une politique mandataire est basée sur une restriction d'accès aux données en utilisant des niveaux de sécurité qui sont attachées aux données et aux sujets. Dans un modèle basé sur les rôles l'accès est limité aux rôles activités

d'un utilisateur. Il propose une alternative au contrôle mandataire très traditionnel et restrictif. Il permet une gestion plus flexible qu'un contrôle discrétionnaire.

Les politiques DAC et RBAC sont basées sur un contrôle structurel (le système vérifie s'il y a adéquation entre une demande d'accès et les règles définies), le MAC est à la fois structurel (une appartenance du sujet et de l'objet à une catégorie) et dynamique (une demande d'accès doit être en accord avec les axiomes en lecture et en écriture).

Les politiques d'administration déterminent qui est autorisé à modifier les autorisations. Dans un contrôle d'accès mandataire les accès permis sont déterminés en se basant sur une classification des sujets et des objets. L'administrateur de sécurité est la seule personne qui peut changer les niveaux de sécurité des sujets et des objets.

Un contrôle d'accès discrétionnaire permet des politiques d'administration diverses : *centralisée, hiérarchique, coopérative, de propriété* ou *décentralisée*.

Les modèles mandataires sont plus robustes (i.e. contrôle d'accès et contrôle de flux) par rapport aux modèles discrétionnaires, mais on peut leur reprocher l'obligation d'adopter une structure rigide en couche pour le système d'information.

Les modèles discrétionnaires et les modèles de rôles n'imposent aucune organisation de l'information. Ils sont adaptés à différents modèles de management de l'information ou à différentes applications.

La proposition RBAC offre des concepts pour faciliter l'administration d'un schéma d'accès. Ces qualités peuvent être utilisées pour mettre en place la sécurité du système d'information.

C'est pourquoi nous pensons que ce modèle répond bien aux besoins de sécurité des systèmes d'information de l'entreprise et aux contraintes d'administration de celui-ci. Nous avons donc privilégié ce modèle dans la suite de nos travaux.

Nous avons présenté dans la dernière partie du chapitre des travaux de mise en oeuvre basée sur le concept de rôle. Elles déterminent plus ou moins bien le processus d'ingénierie des rôles par la définition des droits d'accès attachés aux rôles.

Les travaux les plus proches des notres sont [TOB98] et [FH97]. Le premier présente la décomposition des tâches du concepteur et des tâches de l'administrateur de sécurité pendant la conception et l'exploitation de sécurité d'un système d'information. Le deuxième définit les rôles en utilisant le concept de cas d'utilisation.

Le chapitre suivant présente notre proposition en matière d'ingénierie des rôles d'un système d'information. Il montre la prise en charge de la sécurité logique au niveau de la conception du système d'information avec le langage UML comme le langage de modélisation orientée objet.

Chapitre 3

Un modèle RBAC étendu et sa représentation en UML

Objectifs du chapitre :
- *Introduire une extension au modèle RBAC*
- *Définir le partage des responsabilités dans le processus de réalisation de sécurité d'un système d'information*
- *Présenter le langage de modélisation objet UML*
- *Rapprocher les concepts du modèle RBAC avec certains concepts UML*
- *Développer le processus de création des rôles dans un système d'information*

Dans la dernière partie du chapitre précédent nous avons montré l'insuffisance des méthodes actuelles pour prendre en compte la conception et la mise en place des rôles dans les modèles RBAC.

Notre objectif dans cette thèse est de fournir, dans un premier temps, un moyen pour identifier et spécifier au travers d'une description UML les différentes rôles nécessaires au système d'information. Puis dans un second temps nous voulons générer de façon automatique l'ensemble des rôles et leurs permissions associées dans un formalisme XML indépendant du système informatique cible et du modèle RBAC choisi (Chapitre 5).

Pour cela nous voulons intégrer l'aspect sécurité (i.e. rôles) au niveau de la conception du système d'information. Ce chapitre présente la construction des rôles d'un système d'information pendant la conception de celui-ci. On se basera sur une modélisation orientée objet en utilisant comme support le langage de modélisation UML [Gro97, Gro00a, Mul98].

Dans les bases de données, le contrôle d'accès consiste à protéger les données des consultations et des modifications non autorisées ou incorrectes. Dans notre approche, les utilisateurs accèdent aux données persistantes indirectement au travers des différentes applications (i.e. services) du système d'information et ceci en accord avec les règles définies dans le système de gestion de contrôle d'accès (Figure 3.1). C'est donc au niveau de l'accès aux services que les contrôles se feront.

Nous présentons au début de ce chapitre les différentes structures organisationnelles d'une entreprise et nous montrons comment la notion de rôle s'adapte à celles-ci. Ensuite, nous introduisons une extension du modèle de Sandhu qui permettra plus de flexibilité dans l'organisation des rôles de responsabilité ou de compétence dans l'entreprise. La deuxième partie positionne UML dans le champ des systèmes d'information. La partie suivante propose un rapprochement entre certains concepts UML et certains concepts du modèle RBAC. Puis, nous proposons une descrip-

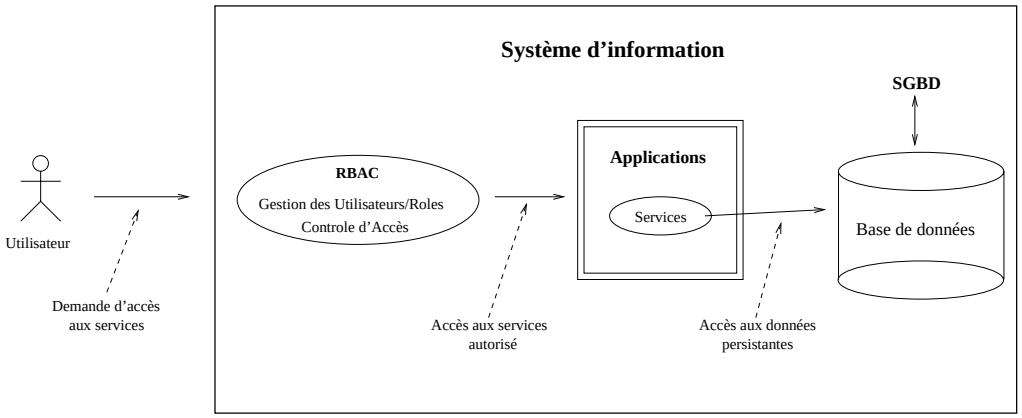


FIG. 3.1 – L'accès aux données par les applications du système d'information.

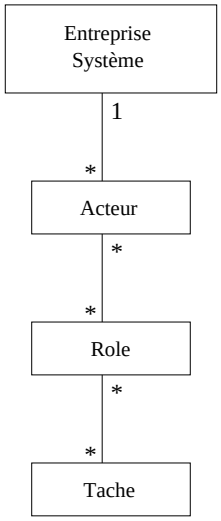


FIG. 3.2 – La vue organisationnelle d'une entreprise.

tion UML naturelle des rôles et le processus de création des rôles dans un système d'information d'une entreprise.

3.1 La structure organisationnelle d'une entreprise

Nous avons présenté dans le Chapitre 1 la structure d'une entreprise, du point de vue fonctionnelle et organique. On peut compléter cette présentation par le point de vue organisationnelle.

L'organisation qualifie tout ce qui a trait à la structure de l'entreprise et donc aux relations entre les éléments qui la composent. L'organisation doit donc décrire :
- les responsabilités et l'autorité des acteurs de l'entreprise,
- les relations entre les membres et les services de l'entreprise,
- le regroupement des diverses activités,
- les tâches à réaliser et l'affectation des individus aux tâches.

Le problème de la répartition des tâches, des responsabilités et des activités entre les membres d'une organisation ou plus généralement de l'élaboration de la structure organisationnelle met

en jeu un très grand nombre de variables aussi diverses que les objectifs, le type de produits, la personnalité des membres de l'entreprise, les modes de contrôle, l'état de la technologie pour n'en citer que quelques-unes [FLLP91]. La notion de rôle va permettre de regrouper les tâches qu'une ou plusieurs personnes doivent assurer.

La Figure 3.2 présente les éléments caractéristiques de la structure organisationnelle d'une entreprise.

La structure organisationnelle d'une entreprise est donc le résultat d'un compromis à trouver entre les objectifs de l'entreprise et les coûts économiques et sociaux mis en oeuvre. Cette conception est essentiellement pragmatique, elle est basée sur l'expérience et la compétence de ceux qui sont chargés de la définir. Chaque entreprise, ayant sa propre spécificité, aura sa propre structure.

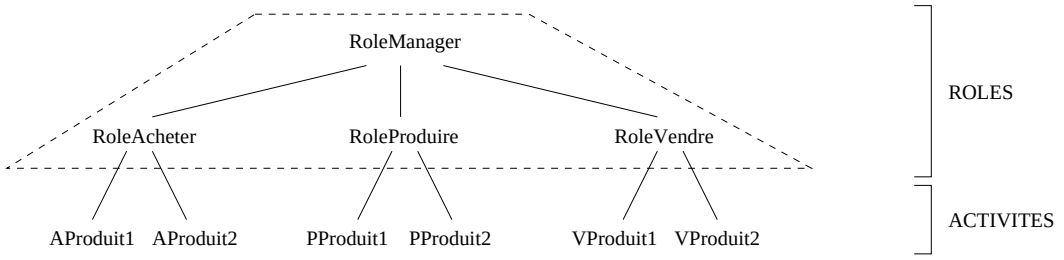
Les structures d'organisation les plus connues sont [FLLP91] :

1. *Structure hiérarchique* optimise les liaisons hiérarchiques. Chaque poste a le pouvoir d'accomplir des actions déterminées et dépend d'un poste hiérarchiquement supérieur. On peut trouver les structures hiérarchiques suivantes :
   (a) la forme solaire ou autocratique,
   (b) la forme hiérarchique "pyramidale".
2. *Structure fonctionnelle* - optimise l'utilisation des compétences ; une organisation est dotée d'une structure fonctionnelle lorsque les unités sont constituées par nature du travail ou par fonction. Les fonctions de l'entreprise peuvent être divisées en sous-fonctions. Les fonctions de l'entreprise sont divisées en sous-fonctions. Cela conduit à une structure horizontale.
3. *Structure divisionnelle* - optimise la réalisation des programmes de production et/ou vente ; le principe consiste en un regroupement des activités de l'entreprise par ensemble de produits-marchés. Chaque division ainsi constituée, est placée sous l'autorité d'un Directeur de Division, responsable des décisions à la fois stratégiques, administratives et opérationnelles. Cela conduit à une structure verticale.
4. *Structure matricielle* - le principe consiste à croiser les structures fonctionnelle et divisionnelles. Cette structure est conçue pour mettre en oeuvre des projets complexes et diversifiés. Chaque groupe de projet ou de produit dispose d'un responsable et d'une équipe de spécialistes assurant la responsabilité complète en termes de planification, de réalisation et de contrôle. Les opérationnels (les membres du groupe) ont une double dépendance, vis-à-vis d'un supérieur hiérarchique au sein de leur fonction et vis-à-vis du responsable du projet dans lequel ils interviennent.

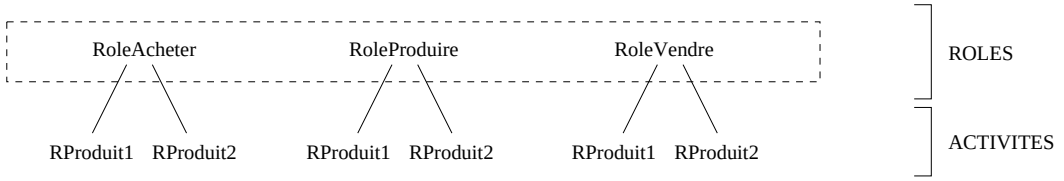
La Figure 3.3 présente les différentes structures d'organisation d'une entreprise décrites ci-dessus. Elle montre une répartition possible de responsabilités et de compétences (i.e. fonctions) dans les différents types de structures à l'aide de la notion de rôle. Dans le cas d'une structure matricielle un partage de responsabilités entre deux rôles est nécessaire.

3.2 Les extensions proposées du modèle RBAC

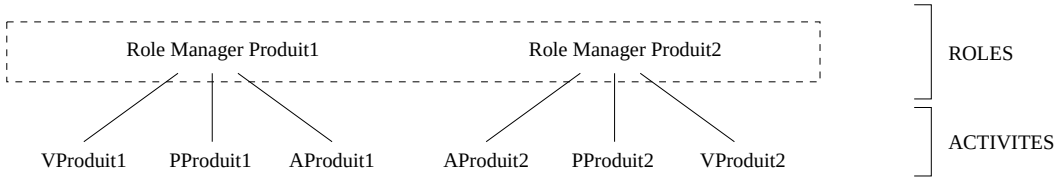
Nous nous basons sur le modèle RBAC proposé par R. S. Sandhu [SCFY96, SS94] pour la gestion des rôles dans une organisation d'entreprise. La complexité des organisations actuelles (i.e. structure matricielle) nous a conduit à proposer pour la gestion des rôles une extension du modèle de Sandhu permettant une décomposition plus fine des responsabilités dans l'entreprise. Nous introduisons pour cela la notion de fonction. Cette décomposition est également présente dans l'approche OSSAD (Chapitre 1) avec les niveaux fonctions et activités.



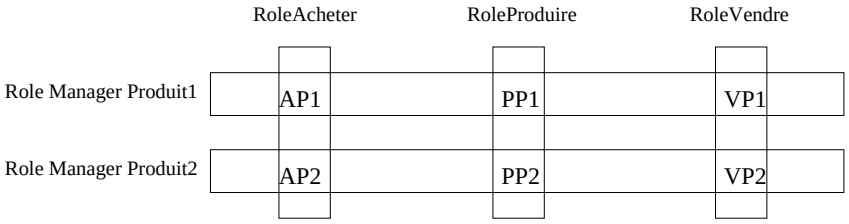
(a) Structure hierarchique de roles



(b) Structure fonctionnelle de roles



(c) Structure divisionnelle de roles



(d) Structure matricielle de roles

AProduit1, AP1 − Achat Produit 1
AProduit2, AP2 − Achat Produit 2

PProduit1, PP1 − Production Produit 1
PProduit2, PP2 − Production Produit 2

VProduit1, VP1 − Vente Produit 1
VProduit2, VP2 − Vente Produit 2

FIG. 3.3 – Les différentes structures d'organisation d'une entreprise.

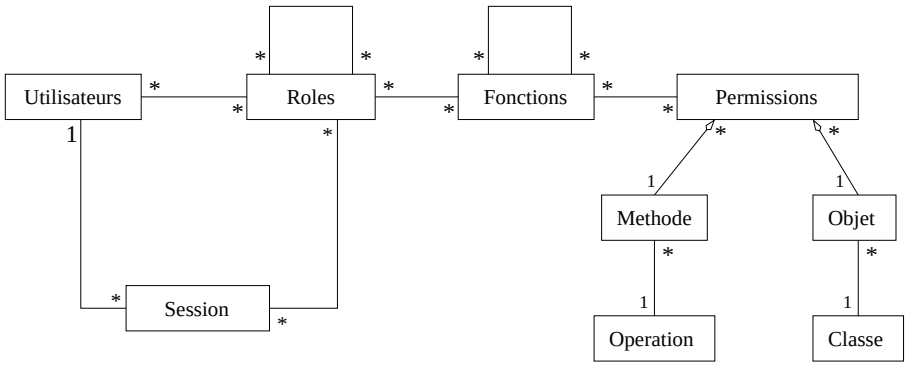


FIG. 3.4 – Une extension du modèle RBAC.

3.2.1 La présentation de l'extension du modèle RBAC classique

Nous spécifions ci-dessus les différents concepts de l'extension du modèle RBAC proposé par R. S. Sandhu. Notre proposition de l'extension du modèle RRBAC est donnée dans la Figure 3.4 sous la forme d'un diagramme de classe d'UML [GH00b, PGH01, GH00c].

Une **session** est la période du temps pendant laquelle une personne autorisée est connectée au système d'information et peut réaliser les activités accessibles pour elle.

Un **utilisateur** est une personne qui accède au système pendant une session ; il peut jouer les rôles qui lui ont été affectés par un administrateur du système**.**

Un utilisateur est donc une personne physique qui manipule les objets du système. Il est défini par le nom *uNom* :

$$u_i \ = \ (uNom_i)$$

$$\forall u_i \in U \ et \ U \text{ - } ensemble \ de \ tous \ les \ utilisateurs$$

Un utilisateur peut avoir plusieurs responsabilités dans une entreprise. Quand un utilisateur se connecte au système il ouvre une session de travail. L'utilisateur a la possibilité d'activer pendant la session un ou plusieurs rôles pour lesquels il a été préalablement déclaré membre.

Une session représente donc une collection de rôles pour un utilisateur actif. Un rôle est activé s'il appartient à une session et un utilisateur est actif s'il appartient à une session.

Chaque rôle qu'on définit dans le modèle RBAC permet de réaliser une tâche spécifique dans un processus de l'entreprise. Un ensemble de rôles détermine les responsabilités d'un poste du travail dans une entreprise.

D'autre part un rôle comprend généralement plusieurs fonctions qu'un utilisateur pourra exercer. On pourra ainsi choisir pour chaque rôle les fonctionnalités du système qui lui seront nécessaires à l'exercice de celui-ci. Nous avons donc introduit ce nouveau concept dans le modèle RBAC.

Un **rôle** est vu donc comme une collection de fonctions. Il est défini par un nom *rNom* et par un ensemble de fonctions *rFon* :

$$r_i \ = \ (rNom_i, rFon_i)$$

$$\forall r_i \in R \ et \ R \text{ - } ensemble \ de \ tous \ les \ rôles$$

Ce niveau supplémentaire permet d'apporter plus de flexibilité dans la gestion de la sécurité et le découpage des responsabilités. Par exemple, si on désire mettre en place une nouvelle organisation dans une entreprise ou une nouvelle application dans un système d'information, ceci provoque généralement la nécessité de redistribuer les différentes fonctions aux acteurs. Dans notre cas,

l'administrateur de sécurité devra seulement modifier les affectations des fonctions aux rôles en tenant compte de la réorganisation. Les utilisateurs auront toujours les mêmes rôles mais avec des fonctions nouvellement attachées. De même, si une fonction doit évoluer dans l'entreprise, on pourra changer l'affectation des permissions à celle-ci sans devoir changer les rôles qui utilisent cette fonction.

Pour cela, nous avons introduit trois relations supplémentaires dans le modèle RBAC :
- la relation R-F qui associe les fonctions aux rôles,
- la relation F-P qui réalise une association des permissions aux fonctions et
- la relation F-F qui détermine la relation d'héritage entre les fonctions.

Dans notre approche, une ou plusieurs permissions peuvent être associées à chaque fonction.

Une **fonction** est donc un ensemble de permissions et elle est définie par un nom *fNom* et par un ensemble de permissions *fPerm* :

$$f_i = (fNom_i, fPerm_i)$$

$$\forall f_i \in F \text{ et } F \text{ - ensemble de toutes les fonctions}$$

On peut aussi définir une relation d'héritage entre les fonctions - la relation F-F. Cette relation indique que la fonction qui hérite d'une autre fonction possède toutes les permissions de la fonction père.

Il faut avoir des droits d'accès (i.e. autorisations) spécifiques pour exercer une fonction particulière d'un rôle. Ces droits déterminent la possibilité d'exécuter une application ou d'accéder à des données particulières.

Dans la suite de ce travail, nous nous basons sur un système d'information orienté objet où les propriétés de chaque objet ne sont accessibles qu'aux travers de méthodes spécifiques, comme dans le modèle IRIS [AA92]. L'hypothèse d'un monde fermé permettra également de ne gérer que des autorisations positives. Si aucune précision sur la méthode n'est donnée alors l'exécution est interdite.

Une permission représente une autorisation d'accès d'un type particulier sur un objet ou un ensemble d'objets protégés par le système de sécurité. Chaque attribut ou service d'un objet n'est accessible de l'extérieur qu'en envoyant un message à l'objet qui le contient. La réception du message déclenche alors l'exécution de la méthode correspondante.

Une permission détermine donc le droit d'exécuter une méthode particulière sur un objet particulier [BS97]. Une **permission** est définie par une méthode *pMethode* et par un objet *pObjet* :

$$p_i = (pMethode_i, pObjet_i)$$

$$\forall p_i \in P \text{ et } P \text{ - ensemble de toutes les permissions}$$

Cette permission est supposée donnée pour toutes les instances d'une classe objet sauf spécification contraire (Chapitre 4). Parfois on doit en effet restreindre cette permission qu'aux seules instances des classes objet satisfaisant une contrainte particulière. Une telle contrainte précise alors que la permission est valide seulement pour un sous-ensemble des instances de la classe objet.

Un exemple de décomposition de rôles peut être pris dans le système d'information d'une université. Un utilisateur *Professeur* joue deux rôles : un rôle d'*Enseignant* et un rôle de *Chercheur*. Le rôle d'Enseignant possède les fonctions : trouver des informations, préparer des cours, donner des cours, préparer des examens, éditer des résultats, modifier des résultats, etc. Le rôle de Chercheur contient les fonctions : trouver des informations, créer une théorie, tester la théorie, documenter des résultats, etc. Des ensembles de permissions sont attachés à ces fonctions pour garantir les accès nécessaires à chaque fonction. La Figure 3.5 présente les relations entre ces éléments.

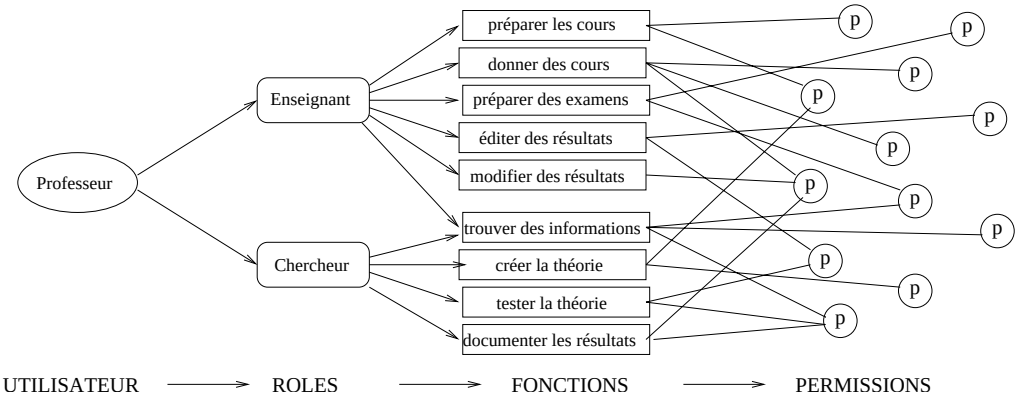


FIG. 3.5 – Un exemple des relations rôle - fonction - permission.

Une **méthode** représente une action simple réalisée dans un système. Elle est définie par un nom *mNom* :

$$m_i = (mNom_i)$$

$$\forall m_i \in M \text{ et } M \text{ - ensembles de toutes les méthodes}$$

Un **objet** est une entité (une ressource) sécurisée du système. Il est défini par un nom *oNom* :

$$o_i = (oNom_i)$$

$$\forall o_i \in O \text{ et } O \text{ - ensembles de tous les objets}$$

Le mode d'accès dépend de l'environnement cible à protéger. Par exemple dans les systèmes classiques de gestion de fichiers l'ensemble des modes est constitué par *read*, *write*, *execute* ; dans un environnement objet un mode d'accès est défini pour chaque méthode publique d'une classe.

On peut ainsi donner une définition formelle d'une permission :

Une permission est définie par une fonction $p(m, o)$ où o est l'ensemble des instances de la classe objet, m est une méthode qui peut être exécutée sur les instances limitées éventuellement par une contrainte.

Une description des contraintes sera présentée dans le chapitre suivant.

Une méthode est une instance particulière d'une **opération**. Chaque opération peut avoir une ou plusieurs méthodes qui différent par exemple sur le nombre d'arguments.

Un objet est une instance d'une **classe** particulière et chaque classe peut avoir plusieurs instances.

La Figure 3.6 présente les classes de base du modèle RBAC. Elles représentent des éléments du système d'information, par exemple la classe *utilisateur* est spécialisée en une classe utilisateur régulier, une classe concepteur et une classe administrateur de sécurité. Ce découpage concerne le partage de responsabilités au niveau du concepteur de l'application, une partie des responsabilités au niveau de l'administrateur de sécurité et enfin au niveau d'un utilisateur régulier qui utilise ce système. Le concepteur et l'administrateur de sécurité du système d'information peuvent spécifier des contraintes sur les différents éléments du modèle.

La classe *rôle* peut être spécialisée en une classe de rôle d'utilisateur et une classe de rôle administratif.

Il existe pour l'administrateur de sécurité des permissions particulières qui lui permettent de modifier les ensembles U, R, F, P et leurs relations - les permissions administratives. Elles sont en relations avec les fonctions et les rôles administratifs.

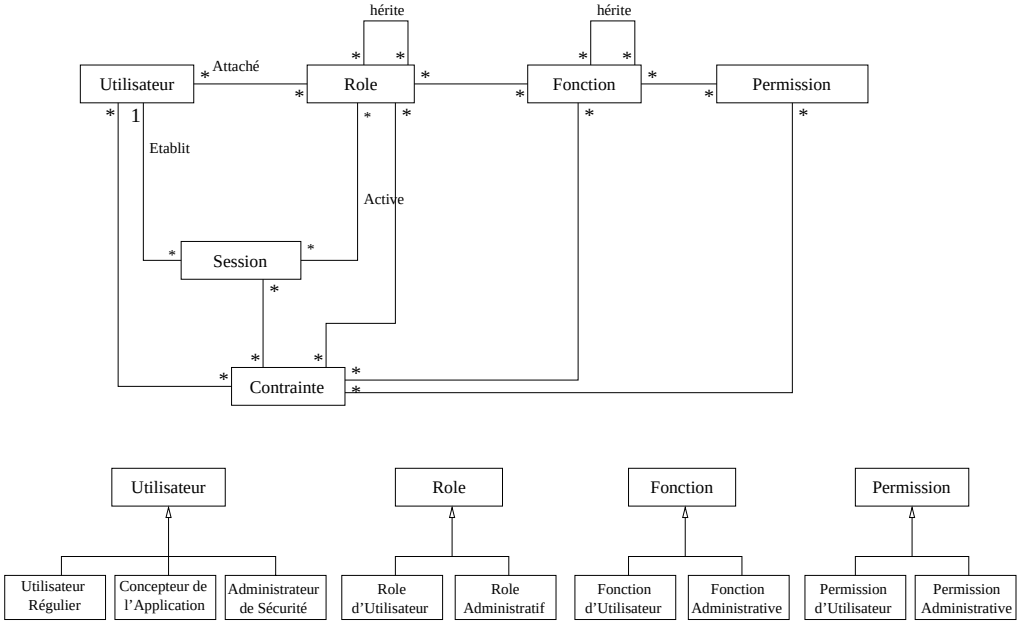


FIG. 3.6 – Les classes d'entités du modèle RBAC de conception.

3.2.2 Le partage des responsabilités

Nous avons choisi d'identifier deux étapes principales dans le processus de réalisation (i.e. définition et exploitation) des rôles d'un système d'information :

- lors de la *conception du système d'information*, le concepteur de l'application détermine les rôles de façon quasi automatique,
- lors de l'*exploitation du système d'information*, l'administrateur de sécurité connaît le profil des utilisateurs, c'est-à-dire les différents rôles qui doivent être attachés à chaque utilisateur en fonction de leurs responsabilités dans l'entreprise.

Cette décomposition est présentée dans le diagramme de classes de la Figure 3.7. La méthode présentée dans le Chapitre 2 [TOB98] propose une décomposition analogue, mais répartie sur sept niveaux d'abstraction.

Dans le processus de développement d'un système, le client demande au concepteur de réaliser une application. Le concepteur définit les éléments de cette application qui sont tirés du cahier des charges.

Ces éléments sont ensuite donnés à l'administrateur de sécurité. Il définit des contraintes qui dépendent de la politique globale de sécurité dans l'entreprise. Les contraintes de l'administrateur doivent être établies en accord avec les contraintes de l'application pour obtenir un système cohérent.

3.2.2.1 L'étape de conception

Cette étape est réalisée par le concepteur d'application. Il définit le modèle de l'application qui contient tous les éléments exprimant les besoins des utilisateurs. Le concepteur génère de ce Modèle l'ensemble des éléments suivants : les rôles, les fonctions, les permissions et les contraintes. Ces ensembles d'éléments doivent être récupérés dans un format qui sera utilisable et lisible pour l'administrateur de sécurité.

Les tâches d'un concepteur d'une application sont :

- l'affectation des éléments : les permissions aux fonctions, les fonctions aux rôles,

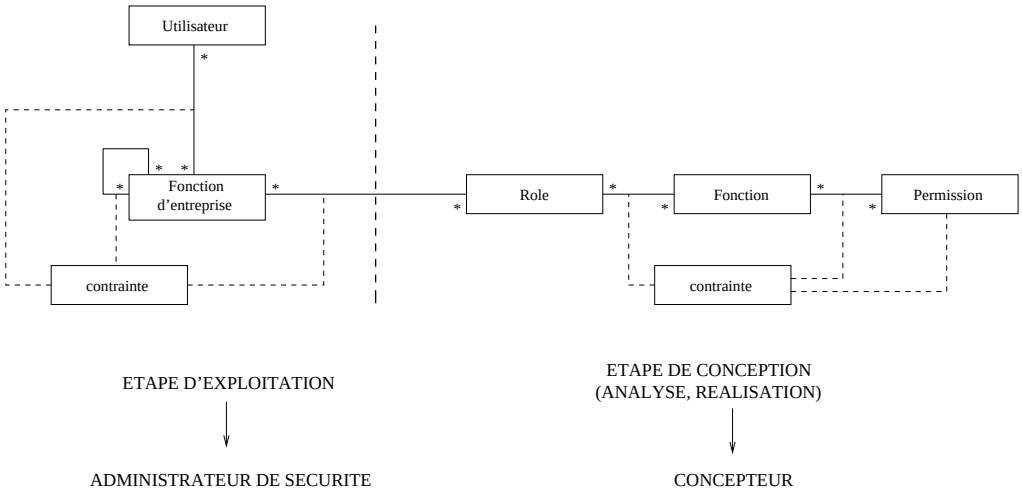


FIG. 3.7 – Les étapes d'un concepteur et d'un administrateur.

– la mise en place des contraintes d'une application.

3.2.2.2 L'étape d'exploitation

L'administrateur de sécurité gère un ensemble d'applications en assurant la cohérence globale du système. Il donne aux utilisateurs du système des droits précis d'utilisation de ces applications. L'administrateur de sécurité doit définir des contraintes sur les affectations (i.e. affectation des utilisateurs aux fonctions d'entreprise et affectation des rôles aux fonctions d'entreprise) qui respectent les contraintes de l'entreprise.

L'administrateur de sécurité obtient du concepteur une application qui contient : un ensemble de rôles, un ensemble de fonctions, un ensemble de permissions et un ensemble de contraintes d'application.
Il dispose, de son coté, de deux groupes d'éléments :
– des personnes qui travaillent dans l'entreprise et
– des fonctions qui sont réalisées dans l'entreprise.

Une ***fonction d'entreprise*** (*fonctionEntreprise*) est une tâche ou un ensemble de tâches qui peuvent être réalisées dans un système par une personne.
L'administrateur de sécurité doit affecter les rôles aux utilisateur par le biais de la relation entre fonction d'entreprise et rôle. On peut définir aussi une hiérarchie de fonctions d'entreprise, alors une fonction d'entreprise peut hériter d'autres fonctions d'entreprise.

Les tâches d'un administrateur de sécurité sont les suivantes :
– l'affectation des éléments : les rôles aux fonctions d'entreprise et les utilisateurs aux fonctions d'entreprise,
– la mise en place des contraintes sur les trois types de relations : utilisateur - fonctionEntreprise, fonctionEntreprise - fonctionEntreprise et fonctionEntreprise - rôle.

3.3 Les concepts du langage UML

On propose dans ce chapitre un "rapprochement" possible entre certains concepts d'UML et ceux du modèle RBAC. Nous rappelons dans cette partie les principaux concepts du langage UML, en insistant plus particulièrement sur ceux que nous utiliserons pour réaliser un modèle RBAC [PG99, PGH01].

3.3.1 La présentation d'UML

UML (*Unified Modeling Language*) est un langage de modélisation graphique pour spécifier, visualiser, construire et documenter les éléments d'un système. Il est devenu le support standard de toute approche de conception orientée objet [Gro97, BRJ98, Gro00a].

UML est un langage pour la représentation des modèles objet, mais le processus d'élaboration de ces modèles n'est pas décrit par UML. Ce n'est pas une méthode de conception, c'est uniquement un langage de description. Les auteurs d'UML insistent sur les caractéristiques suivantes qui leur semblent essentielles pour initier un processus de développement :

- piloté par les cas d'utilisation - toutes les activités, la spécification de ce qui doit être fait jusqu'à la maintenance, sont guidées par les cas d'utilisation,
- centré sur l'architecture - l'architecture est conçue pour satisfaire les besoins exprimés dans les cas d'utilisation, mais aussi pour prendre en compte les évolutions et les contraintes de réalisation et ceci à l'aide de vues.

Les cas d'utilisation viennent en complément de l'approche objet et forment la base du processus de développement. Ils doivent baliser les différentes activités et étapes du processus de développement :

- analyse qui permet de capturer, de clarifier et de valider les cas d'utilisation,
- conception et réalisation des cas d'utilisation,
- vérification que les cas d'utilisation satisfassent tous les besoins d'un utilisateur.

Nous voulons également utiliser ceux-ci pour spécifier le contrôle d'accès du système d'information.

UML présente de plus, plusieurs caractéristiques qui en font un bon outil pour l'élaboration d'un cahier de charge :

- il a fait l'objet d'une normalisation par l'OMG (Object Management Group) qui a adopté UML en 1997,
- il est bien accepté par la communité des informaticiens ; c'est un langage qui facilite le dialogue entre le maître d'ouvrage et le maître d'oeuvre du système d'information (les définitions ci-dessous),
- il sert de support génie logiciel pour réaliser le cycle de développement du système d'information depuis l'expression des besoins jusqu'à la génération de tout ou partie de l'application.

Maître d'ouvrage est le client demandeur du futur système d'information, cette personne exprime les besoins des futurs utilisateurs. D'autres rôles lui sont dévolus tels que la formalisation des objectifs, le choix des moyens, le suivi de la réalisation et le pilotage de l'exploitation.

Maître d'oeuvre est le fournisseur (informaticien) qui réalise le système d'information informatisé. C'est un responsable chargé par le maître d'ouvrage de concevoir techniquement, de conduire et réaliser les travaux, de contrôler les résultats et préparer l'exploitation.

3.3.1.1 Architecture d'UML

Avec le langage UML on peut regarder le système d'information selon des points de vues différents. Plusieurs types de diagrammes permettent de modéliser ceux-ci. Un diagramme est une représentation graphique de tout ou partie d'un modèle.

L'architecture d'UML présente deux éléments (Figure 3.8) : d'une part les constituants et d'autre part les vues. Les constituants d'UML sont :

- les éléments de modélisation qui représentent les propriétés du langage,
- les diagrammes qui constituent les expressions graphiques pour visualiser un modèle.

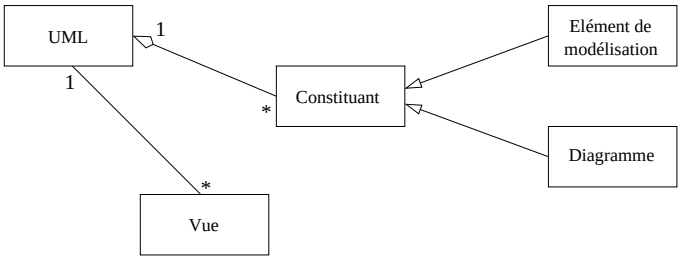


FIG. 3.8 – Les éléments de l'architecture d'UML.

FIG. 3.9 – Un approche de 4+1 vues dans l'architecture d'UML.

3.3.1.2 Les vues

Ph. Kruchten [Kru95, Mul98] a proposé un ensemble de 4+1 vues pour exprimer les différentes perspectives de l'architecture d'un système d'information (Figure 3.9) :

1. *vue logique* - cette vue concerne "l'intégrité de conception", elle décrit les aspects statiques et dynamiques d'un système en termes de classes et d'objets et se concentre sur l'abstraction, l'encapsulation et l'uniformité. Au-delà de la satisfaction des besoins fonctionnels de l'utilisateur, cette vue permet d'identifier et de généraliser les éléments et les mécanismes qui constituent les différentes parties du système.
2. *vue des processus* - cette vue concerne "l'intégrité d'exécution" ; elle représente la décomposition en flots d'exécution, la synchronisation entre flots et l'allocation des objets et des classes au sein des différents flots. Elle prend toute son importance dans les environnements multi-tâches. Elle exprime la perspective sur les activités concurrentes et parallèles (tâches et processus) du système.
3. *vue des composants* - cette vue concerne "l'intégrité de gestion" ; elle se préoccupe de l'organisation des modules dans l'environnement de développement. Elle exprime la perspective physique de l'organisation du code en termes de modules, de composants et de l'environnement d'implémentation.
4. *vue de déploiement* - cette vue concerne "l'intégrité de performance" ; elle décrit les différentes ressources matérielles et la distribution des composants du système d'information sur ces ressources. Elle exprime la répartition du système à travers un réseau de calculateurs et de noeuds logiques de traitement.
5. *vue des cas d'utilisation* unifie les quatre vues précédentes ; elle permet d'identifier les interfaces critiques, elle force les concepteurs à se concentrer sur les problèmes de l'utilisateur, elle guide et valide les autres vues de l'architecture.

3.3.1.3 Les diagrammes

UML définit neuf sortes de diagrammes pour représenter les différents points de vue de modélisation (Figure 3.10). Un diagramme donne à l'utilisateur un moyen de visualiser et de manipuler des éléments de modélisation.

Les diagrammes définis par UML sont [Mul98, Gro00a] :

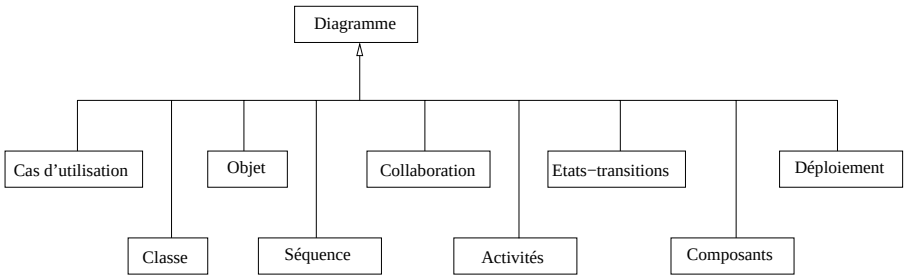


FIG. 3.10 – Les diagrammes définis par UML.

- *diagramme de cas d'utilisation* représente les fonctions du système du point de vue de l'utilisateur,
- *diagramme de classes* représente la structure statique du système d'information en terme de classes et de relations entre ces classes,
- *diagramme d'objets* représente des objets particuliers et leurs relations en un point donné et à un moment donnée,
- *diagramme de collaboration* est une représentation spatiale de la coopération entre les objets du système étudié. Il exprime à la fois le contexte d'un groupe d'objets (au travers des objets et des liens) et l'interaction entre ces objets (par la représentation des envois de messages),
- *diagramme de séquence* est une représentation temporelle de la coopération entre les objets du système étudié. Il montre les interactions arrangées dans une séquence de temps,
- *diagramme d'états-transitions* représente le comportement d'une classe en terme d'états. Il met en évidence l'enchaînement des différents états en fonction des événements externes significatifs,
- *diagramme d'activités* représente le comportement d'un processus en terme d'activités et leurs synchronisations (flots de contrôle),
- *diagramme de composants* représente les composants physiques (modules, programmes, processus, etc.) issus du découpage d'une application,
- *diagramme de déploiement* représente le déploiement des composants sur l'architecture matérielle (support du système d'information).

Les sept premiers diagrammes peuvent aussi bien être utilisés par le maître d'ouvrage que par le maître d'oeuvre. Les deux derniers diagrammes sont utilisés exclusivement par le maître d'oeuvre.

Dans le cadre de notre projet on s'intéressera surtout aux diagrammes de cas d'utilisation et aux diagrammes de séquence qui vont nous permettre de récupérer les permissions (privilèges) et de les associer aux différentes fonctions, rôles et utilisateurs.

Nous détaillons ci-dessous ces deux types de diagrammes.

3.3.2 Diagramme de cas d'utilisation

Un diagramme de cas d'utilisation représente les fonctions du système du point de vue de l'utilisateur. Il détermine le comportement du système sans en avoir à définir ses structures internes.

Ce type de diagramme permet d'identifier les processus de l'entreprise et les acteurs qui participent à ces processus. C'est un diagramme de haut niveau qui représente la vue externe du système (la vue globale du système d'information).

Il introduit les concepts suivants :

- acteur,
- cas d'utilisation et
- relations entre ces éléments - associations entre des acteurs et des cas d'utilisation, ou relations entre cas d'utilisation, ou relations entre acteurs.

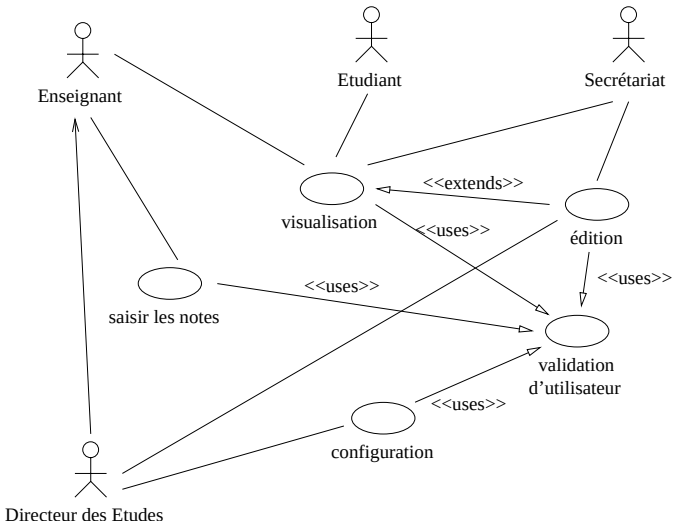


FIG. 3.11 – Un exemple du diagramme de cas d'utilisation "Gestion des Notes".

Un ***acteur*** (actor) représente une interface entre le monde extérieur et le système modélisé. Il représente un rôle joué par une personne qui interagit avec le système. Une personne ou un utilisateur peut jouer plusieurs rôles dans l'entreprise et un même rôle peut être joué par plusieurs utilisateurs. Un acteur interagit avec les cas d'utilisation du système par des envois de messages et des échanges de données [KMPRS98].

Un ***cas d'utilisation*** (use case) est utilisé pour spécifier ou caractériser le comportement d'un système ou d'une application. Un cas d'utilisation représente un point de vue fonctionnel, c'est-à-dire une fonction essentielle du système d'information comme par exemple un processus métier de l'entreprise. Un cas d'utilisation est composé d'une séquence d'actions déclenchée par un acteur externe et qui produit un résultat identifiable. Il définit un scénario du système d'information. La collection des cas d'utilisation spécifie toutes les façons d'utiliser le système.

Il existe plusieurs types de ***relations*** standardisées entre les éléments d'un diagramme de cas d'utilisation [Gro97, Gro00a] :

1. *relation d'héritage* entre les acteurs, au sens des langages orientés objets, permettant de définir des hiérarchies d'acteurs,
2. *relation de communication* - une relation de participation entre un acteur et un cas d'utilisation,
3. *relation d'extension* (<<extends>>) - une relation entre des cas d'utilisation ; elle signifie que le cas d'utilisation source particularise le comportement du cas d'utilisation destination,
4. *relation d'utilisation* (<<uses>>) - une relation entre des cas d'utilisation ; elle signifie qu'une instance du cas d'utilisation source contient (i.e. utilise) également le comportement décrit par le cas d'utilisation destination.

La Figure 3.11 présente un exemple de diagramme de cas d'utilisation regroupant tous les concepts présentés. L'application décrit une gestion des notes au sein d'une faculté. Cette application est présentée dans l'Annexe A. On distingue sur le diagramme les acteurs (Enseignant, Etudiant, Directeur des Etudes, Secrétariat), les cas d'utilisation (visualiser, édition, configuration, saisir les notes, validation d'utilisateur) et les relations entre ces éléments.

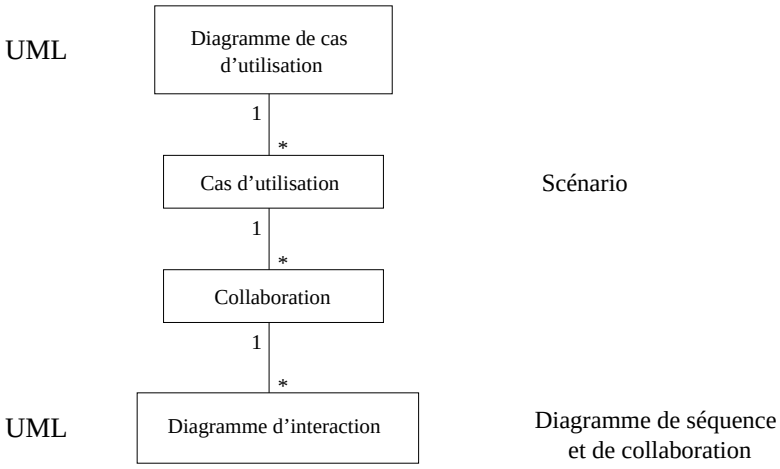


FIG. 3.12 – L'extrait du méta-modèle d'UML.

Le méta-modèle d'UML définit que chaque diagramme de cas d'utilisation peut contenir plusieurs cas d'utilisation et que chaque cas d'utilisation représente une collaboration ou un scénario possible.

Une ***collaboration*** contient :
- la partie structurelle décrite par un ou plusieurs diagrammes de classes et
- la partie comportementale décrite par les diagrammes d'interaction qui représente les aspects dynamiques de la collaboration.

Un diagramme d'interaction permet de décrire chaque collaboration (Figure 3.12).

3.3.3 Diagramme d'interaction

Un diagramme d'interaction capture le comportement d'un seul cas d'utilisation. Il représente les objets et les messages échangées à l'intérieur d'un cas d'utilisation.

Une ***interaction*** est une description de l'envoi de messages entre des instances d'objets pour réaliser une tâche précise. Une interaction décrit le comportement des objets en précisant l'ensemble des messages qu'ils échangent.

Il existe deux types de diagrammes d'interaction :
- diagramme de séquence,
- diagramme de collaboration.

Un **diagramme de séquence** représente une interaction des objets par une séquence des messages dans le temps. Il permet de visualiser la séquence des messages par une lecture de bas en haut. L'axe vertical représente le temps tandis que l'axe horizontal représente les objets qui participent à la collaboration. Une ligne verticale partielle est attachée à chaque objet et représente son cycle de vie.

Un ***objet*** (object) est une entité ayant une frontière qui encapsule un état et un comportement. L'état est représenté par des attributs et des relations, le comportement est représenté par des opérations, des méthodes et des machines à états. Un objet est une instance d'une classe [KMPRS98].

Un ***message*** est une description d'une communication entre des instances d'objets qui correspond à une demande de service à l'objet récepteur par l'objet émetteur. La réception de l'instance du message est normalement considérée comme une instance d'un événement [KMPRS98].

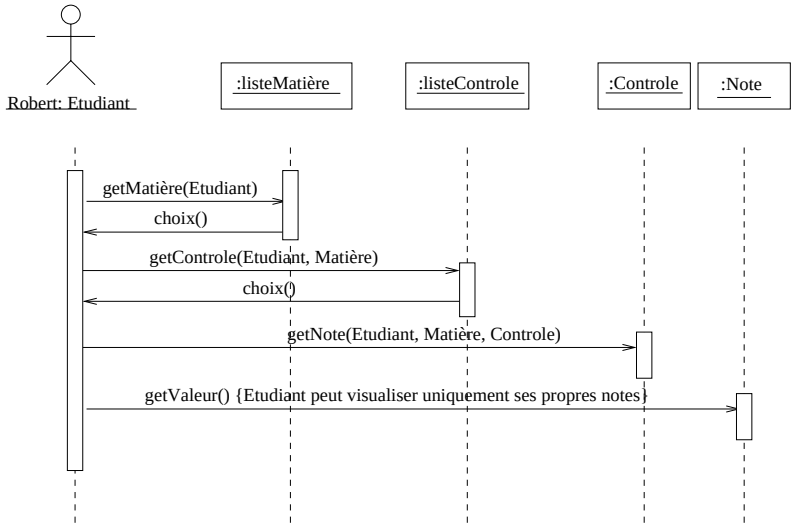


FIG. 3.13 – Un exemple de diagramme de séquence.

La Figure 3.13 présente un exemple de diagramme de séquence. Ce diagramme contient les éléments suivants : un acteur (*Etudiant*), quatre objets (un de la classe *listeMatière*, un de la classe *listeContrôle*, un de la classe *Contrôle* et un de la classe *Note*) et les messages envoyés par l'acteur aux différents objets.

Un **diagramme de collaboration** est une variante du diagramme précédent qui permet de mettre en évidence les interactions entre les différents objets d'un point de vue spatial.

Le diagramme de collaboration de la Figure 3.14 correspond au diagramme de séquence de la Figure 3.13.

3.3.4 Le méta-modèle d'UML

UML est un langage fondé sur un méta-modèle lui-même décrit en UML. Le méta-modèle d'UML définit la sémantique complète pour représenter les modèles d'UML [BRJ98, Res00]. Il est défini comme l'un des quatre niveaux d'une architecture construite en couches [Gro97] :

1. ***meta-metamodel*** - forme la base de l'architecture de méta-modélisation ; il définit un langage pour la spécification de méta-modèle (i.e. couche *metamodel*),
2. ***metamodel*** - est une instance du *meta-metamodel* ; il définit un langage pour la spécification d'un modèle (i.e. couche *model*),
3. ***model*** - est une instance du *metamodel* ; il définit un langage pour décrire un domaine d'information,
4. ***user objets*** - est une instance du *model* ; il définit un domaine d'information spécifique.

Le méta-modèle d'UML est relativement complexe. Il est organisé en paquetages logiques. Ces paquetages regroupent les méta-classes qui ont une forte cohérence entre elles. Un méta-modèle d'UML est décomposé en trois paquetages de haut niveau (Figure 3.15).

Le méta-modèle d'UML que nous utilisons pour recouper les informations nécessaires dans notre processus de production de rôles associés au système d'information est présenté en détails dans l'Annexe B.

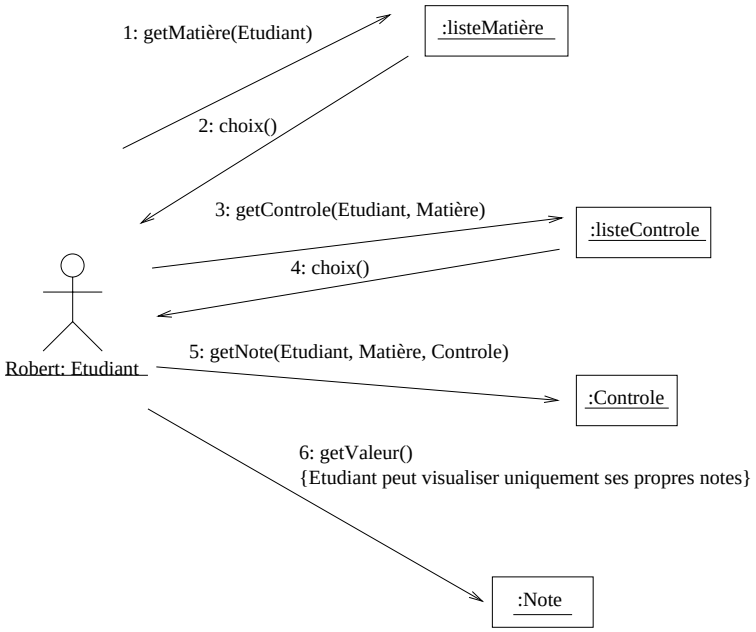


FIG. 3.14 – Un exemple de diagramme de collaboration.

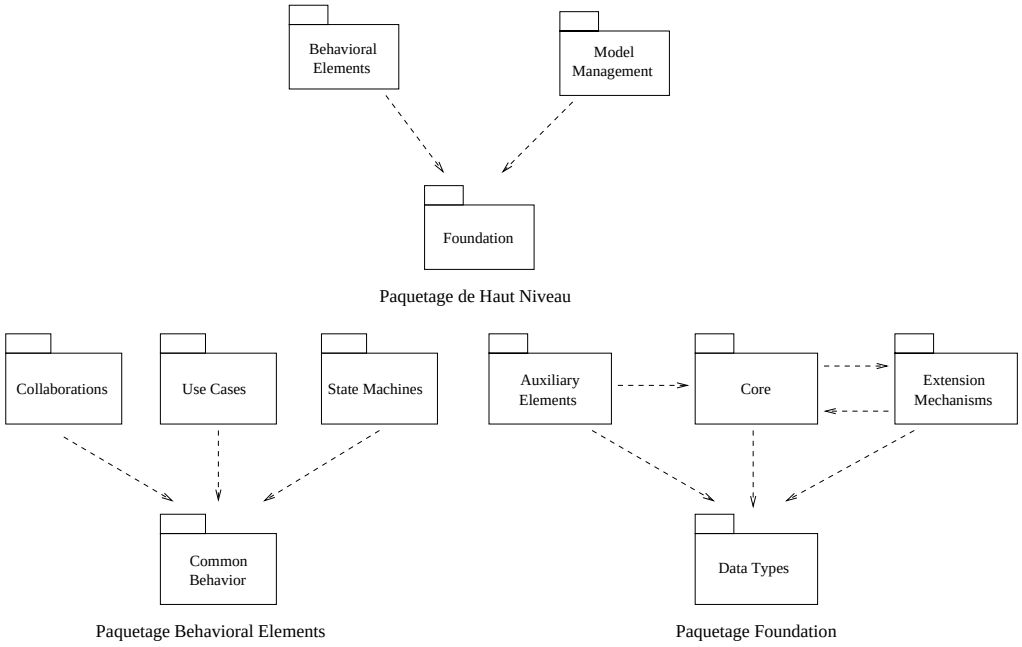


FIG. 3.15 – Les paquetages de haut niveau d'UML.

3.3.5 UML pour la modélisation du système d'information

UML est un langage de modélisation orienté objet. Plusieurs critères nous ont fait choisir UML comme langage de spécification des rôles pour modéliser et implémenter la sécurité du système d'information :

- *prise en compte des besoins de modélisation divers* : UML permet de modéliser la complexité du système d'information grâce à une approche multi-vues (l'approche 4+1 vues),
- *généralité* : UML donne la possibilité de présenter et de modéliser divers types de systèmes du plus petit au plus grand pour tous les domaines d'application,
- *prise en compte de la variété des technologies existantes* : UML peut être utilisé en connexion avec des langages de programmation objet comme Java, Ada, C++, des langages de description comme XML et XMI, des environnements de programmation (e.g. Delphi, PowerBuilder), des technologies de composants (ActiveX, JavaBeans), des technologies de distribution (COM, CORBA), des bases de données (relationnelles, orientées objets),
- *complétude* : UML est un langage complet qui est le résultat de l'unification des concepts de technologies objet, de technologies à base de composants, de modèles de spécification de la dynamique, de langages formels de contraintes (le langage OCL),
- *innovation* : UML permet de modéliser et d'implémenter le modèle RBAC pour gérer la sécurité du système d'information grâce au concept de cas d'utilisation qui constitue une grande avancée à la modélisation des systèmes,
- *possibilité d'automatisation* : un outil comme Rose de Rational Inc. gère automatiquement le code vers les langages de programmation à partir des diagrammes et modèles UML,
- *ouverture* : UML n'impose pas de processus de développement logiciel particulier, il suffit que le processus soit développé à partir des cas d'utilisation.

3.4 Le rapprochement des concepts d'UML et du modèle RBAC

Le système d'information d'une entreprise peut être présenté en UML sous différents modèles qui seront représentés à l'aide de diagrammes d'UML. Le plus important dans la réalisation du modèle RBAC est l'identification d'un ensemble de rôles nécessaires au système d'information.

Pour construire le système de contrôle d'accès de type RBAC, il faut pouvoir déterminer l'ensemble des autorisations qui permettra de créer les rôles du système d'information.

Nous voulons profiter de la phase de création du système d'information pour déterminer au plus tôt les rôles pertinents. Il serait donc intéressant de rapprocher certains concepts du langage UML et ceux du modèle RBAC pour spécifier nos rôles (Figure 3.16) [PGH01, Pon01].

3.4.1 Le modèle RBAC étendu en utilisant le langage UML

Au niveau de la conception d'un système (i.e. première étape) nous prenons uniquement en considération les éléments suivants : rôles, fonctions, permissions, méthodes et objets. Les autres éléments, comme l'utilisateur ou la session seront considérés au niveau de l'exploitation du système.

Dans la phase d'exploitation du système d'information (i.e. deuxième étape), la tâche qui consiste à identifier les utilisateurs et à leur affecter les rôles nécessaires à l'exercice de leur poste, sera réalisée par l'administrateur de sécurité de ce système. L'administrateur s'occupera uniquement des relations U-R dans le modèle RBAC. Nous pensons ainsi, qu'il est possible avec UML de déléguer au niveau du concepteur du système d'information, la conception des rôles [PGH01].

3.4.1.1 Le rôle - l'acteur

Un rôle représente une tâche qui interagit dans un ou plusieurs processus de l'organisation. Un rôle réunit généralement un ensemble de fonctions de l'entreprise et chaque rôle peut être joué par un ou plusieurs utilisateurs ou acteurs de l'entreprise.

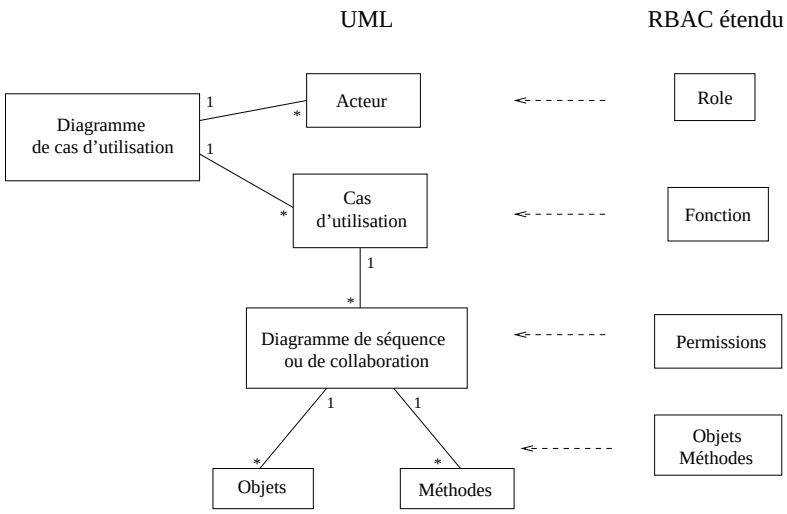


FIG. 3.16 – Du modèle RBAC étendu aux concepts d'UML.

Les auteurs d'UML ont apporté le concept de cas d'utilisation sur lequel le système tout entier peut être développé. Cette conception est réalisée par un diagramme de cas d'utilisation qui utilise entre autres le concept d'acteur. Le concept d'acteur introduit dans les diagrammes de cas d'utilisation est très proche du concept de rôle dans le modèle RBAC. Un acteur est défini en UML comme un ensemble de rôles qu'un utilisateur peut utiliser pour interagir avec le système d'information. Il représente une catégorie d'utilisateurs qui partagent les mêmes fonctions ou les mêmes activités dans une organisation. On peut alors rassembler les utilisateurs du système en groupes spécifiques qui réalisent des fonctions identiques au sein d'une entreprise. Une instance du concept d'acteur est un utilisateur qui interagit avec le système.

Ainsi, dans la phase de conception du système d'information, la notion d'acteur présentée dans les diagrammes de cas d'utilisation nous permettra d'identifier les rôles nécessaires du modèle RBAC. Cet ensemble de rôles pourra être identifié et généré automatiquement (Chapitre 5).

3.4.1.2 La fonction - le cas d'utilisation

Un rôle est vu comme un ensemble cohérent de fonctions qui seront matérialisées dans le langage UML par les cas d'utilisations.

Une fonction dans notre modèle RBAC étendu est un ensemble d'actions exécutées pour réaliser une tâche précise dans un système. Un cas d'utilisation en UML est présenté par une séquence d'appels de méthodes dans un diagramme d'interaction.

Donc, on peut représenter une fonction du modèle RBAC par un cas d'utilisation en UML.

3.4.1.3 Les méthodes et les objets

Les méthodes du modèle RBAC sont représentées en UML par les méthodes exécutées dans les différents diagrammes, e.g. diagramme de séquence et les objets du modèle RBAC par les objets d'UML.

3.4.1.4 Les permissions - le diagramme d'interaction

Un cas d'utilisation représente une collaboration entre les objets qui est décrite à l'aide de diagrammes d'interactions - les diagrammes de séquence ou les diagrammes de collaboration. Chaque cas d'utilisation peut être décrit par une ou plusieurs collaborations et une collaboration est décrite par une ou plusieurs interactions.

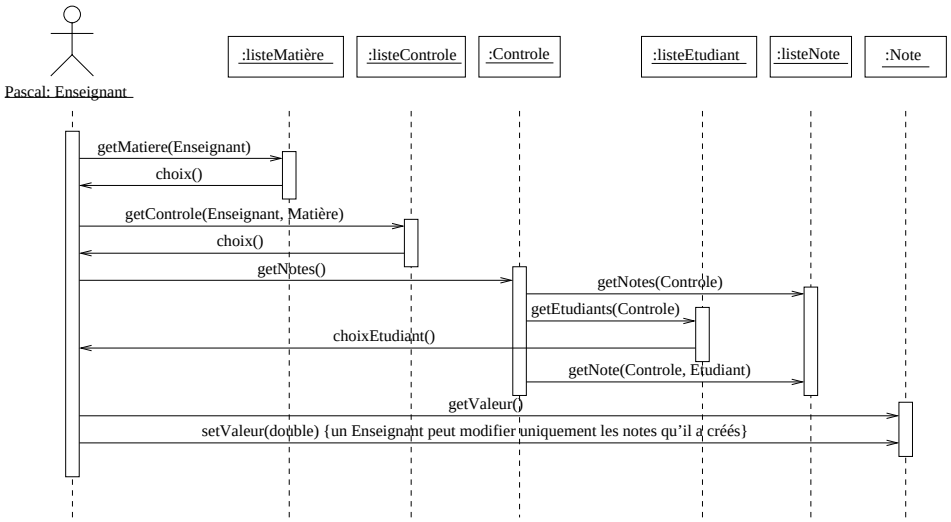


FIG. 3.17 – Un exemple du diagramme de séquence "Modification des Notes".

Dans nos travaux nous avons choisi d'utiliser les diagrammes de séquence pour réaliser des cas d'utilisation du système mais la même démarche pourrait être faite avec les diagrammes de collaboration.

Ce type de diagramme représente une interaction des objets par une séquence de messages échangées entre les objets. Ainsi les acteurs participent directement au processus d'envoi des messages entre les objets. Ce processus est réalisé par l'exécution de différentes méthodes sur les objets recevant les messages (Figure 3.17).

Un message envoyé à un objet récepteur provoque l'activation d'une méthode particulière. L'accès à un attribut ou service d'un objet doit être contrôlé par rapport au droit d'exécution de la méthode correspondante que doit posséder le sujet émetteur du message sur cet objet. Le contrôle d'accès consistera à exprimer pour chaque sujet l'ensemble des méthodes qu'il est autorisé à exécuter.

On peut ainsi déterminer toutes les permissions nécessaires à l'exécution de méthodes correspondantes au cas en recherchant l'ensemble des messages émis directement par un acteur. Ces permissions sont en réalité associées à la fonction décrite par le cas d'utilisation.

Nous avons défini une permission comme un couple : (*methode*, *objet*). La permission indique le doit qu'une méthode peut être exécutée sur un objet.

Le parcours d'un diagramme de séquence retourne un ensemble de permissions - un ensemble de couples (*methode*, *objet*) - associées au cas d'utilisation représenté par ce diagramme de séquence.

3.4.1.5 Les contraintes

On rapproche le concept de contrainte du modèle RBAC avec celle-ci du langage UML :

SI *dans le diagramme de séquence il n'y a aucune contrainte sur le droit d'exécution d'une méthode,*

ALORS *le droit d'exécution de la méthode concerne toutes les instances de la classe de cette méthode.*

Par exemple le diagramme présenté dans la Figure 3.17 définit le comportement du cas d'utilisation "Modification des Notes". Un *Enseignant* peut créer ou modifier les *Notes* d'un *Contrôle* qu'il aura créées au préalable.

Si on veut restreindre la permission d'exécution d'une méthode à certaines instances de la classe, on peut définir des contraintes associées à la méthode. Un message *setValeur(double)* a une

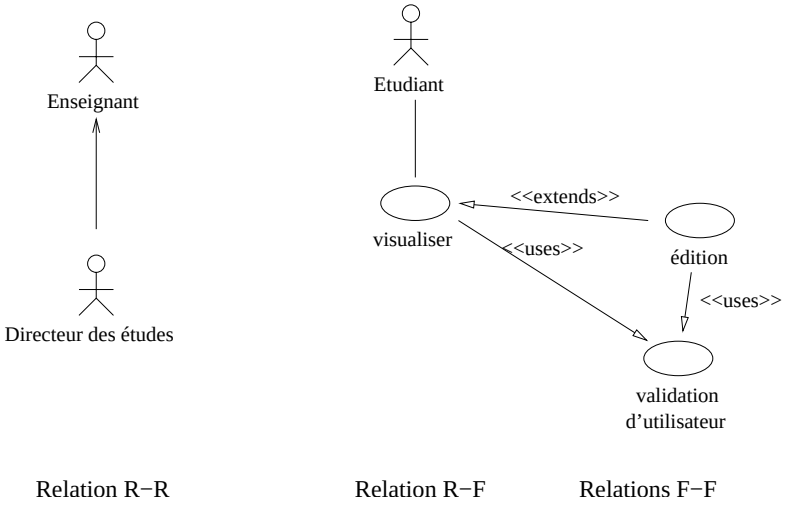


FIG. 3.18 – Les relations entre les éléments.

contrainte comme *{un Enseignant peut modifier seulement les notes qu'il a créées}*, nous précise que seul l'instance de l'*Enseignant* qui enseigne cette *Matière* a le droit d'exécuter cette opération.

La présentation des contraintes est décrite dans le chapitre suivant.

3.4.1.6 Les relations

Les différentes relations entre les éléments du modèle RBAC peut être retrouvées dans les diagrammes de cas d'utilisation et les diagrammes d'interaction (i.e. diagrammes de séquence).

On va présenter la représentation de chaque type de relations du modèle RBAC en utilisant les concepts d'UML.

Relation R-F

Les relations entre des rôles et des fonctions (relations R-F) sont représentées par les relations de communication, autrement dit les relations de participation entre des acteurs et des cas d'utilisation dans les diagrammes de cas d'utilisation (Figure 3.18).

Relation R-R

Une hiérarchie de rôles - une relation R-R (relation entre des rôles) peut être représentée dans un diagramme de cas d'utilisation par une relation de généralisation entre des acteurs appropriés. Cette solution donne la possibilité qu'un acteur (rôle parent) puisse réaliser des fonctions (représentées par des cas d'utilisation) attachées à un autre acteur (rôle fils). Ainsi, un acteur parent possédera les mêmes permissions que l'acteur fils.

La Figure 3.18 présente un exemple d'une relation d'héritage entre deux acteurs (rôles). Un acteur *Directeur des Etudes* représente un acteur père et il hérite d'un acteur fils *Enseignant* : l'acteur *Directeur des Etudes* aura les mêmes privilèges que l'acteur *Enseignant*. On devra ajouter les privilèges associés à *Enseignant* à l'ensemble des privilèges du *Directeur des Etudes*.

Dans ce cas, un acteur spécialisé est en relation avec l'ensemble des cas d'utilisation de l'acteur plus général. Ce type de relation permet de modéliser plus facilement la structure réelle de la hiérarchie d'une organisation.

Relation F-F

Les relations entre les fonctions (relations F-F) sont représentées par les relations entre les cas d'utilisation dans les diagrammes de cas d'utilisation (Figure 3.18).

Il existe deux types de relations entre les cas d'utilisation :

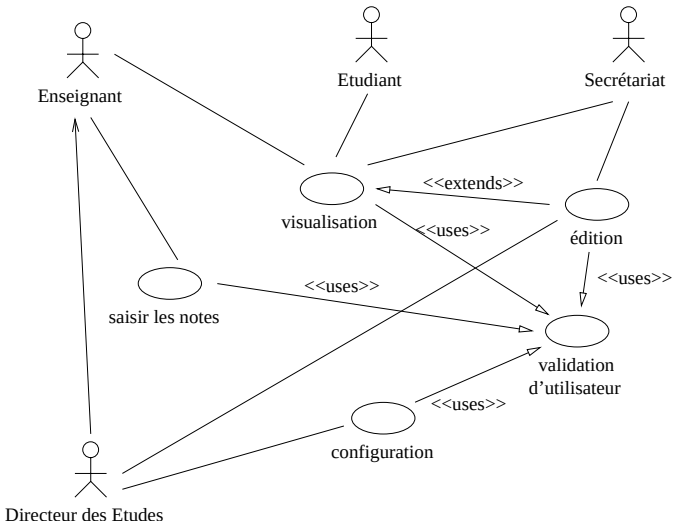


FIG. 3.19 – Le diagramme de cas d'utilisation "Gestion des Notes".

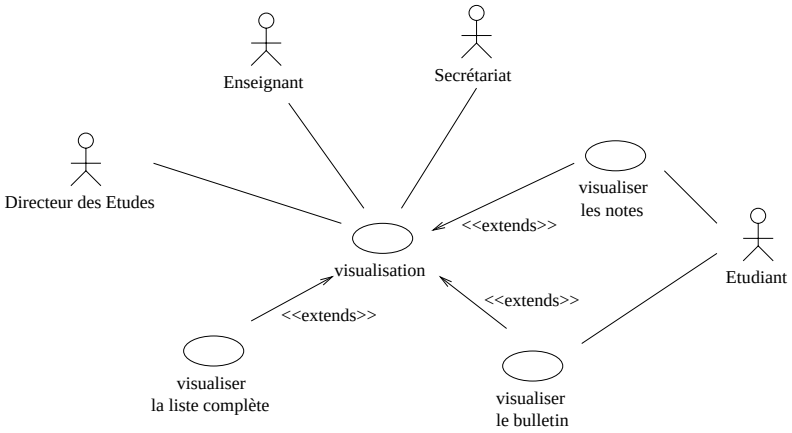


FIG. 3.20 – Le diagramme de cas d'utilisation "Visualisation".

- ***relation d'extension*** (<<extends>>) - relation qui complète un cas d'utilisation particulier ; le cas d'utilisation est étendu par les autres cas d'utilisation.
  La Figure 3.20 présente un exemple dans lequel les acteurs *Directeur des Etudes*, *Enseignant* et *Secrétariat* peuvent réaliser les cas d'utilisation *visualiser la liste complète*, *visualiser le bulletin* et *visualiser les notes* en interagissant avec le cas d'utilisation *visualiser*. L'acteur *Etudiant* peut seulement *visualiser le bulletin* et *visualiser les notes*. L'ajout de l'ensemble des privilèges de *visualiser le bulletin* et *visualiser les notes* à l'ensemble des privilèges de *visualiser* permettrait aux acteurs *Directeur des Etudes*, *Enseignant* et *Secrétariat* de faire une visualisation selon les deux types, ce qui est convenable.
- ***relation d'utilisation*** (<<uses>>) - relation qui donne la possibilité d'utiliser un cas d'utilisation particulier par les autres cas d'utilisation ; les acteurs du cas d'utilisation qui utilise les autres cas d'utilisation peuvent aussi utiliser leurs privilèges.
  Par exemple dans le diagramme de cas d'utilisation présenté dans la Figure 3.19 il existe la relation d'utilisation entre le cas d'utilisation *Configuration* et le cas d'utilisation *Validation d'utilisateur*.

Relation F-P
Les relations entre des fonctions et des permissions (relations F-P) sont déterminées par les liaisons entre les cas d'utilisation et leurs diagrammes d'interaction (i.e. diagrammes de séquence), qui décrivent ces cas d'utilisation.

3.4.2 L'implémentation du modèle RBAC - la création des rôles

En se basant sur le rapprochement présenté ci-dessus, on propose une spécification du contrôle d'accès (type RBAC) du système de sécurité au niveau conceptuel en utilisant comme support le langage de modélisation UML.

Nous utilisons le méta-modèle du langage UML pour définir les rôles du modèle RBAC, les fonctions qui seront utilisées par les rôles pour interagir avec le système d'information et les permissions nécessaires pour l'exécution de ces fonctions. Les diagrammes de cas d'utilisation donnent la liste des acteurs qui interagissent avec le système d'information. L'analyse des diagrammes de cas d'utilisation permet de définir de manière automatique les relations R-R (en utilisant les relations de généralisation entre les acteurs), les relations R-F (en utilisant les relations d'associations entre les acteurs et les cas d'utilisations) et les relations F-F (en utilisant les relations d'extension et/ou d'utilisation entre les cas d'utilisation).
La description d'un cas d'utilisation par les diagrammes d'interaction (en particulier par les diagrammes de séquence) indique les messages qui seront nécessaires pour réaliser les fonctions du système d'information. Chaque message déclenche l'exécution d'une méthode dans l'objet qui le reçoit. Nous pouvons ajouter la permission *exécute* pour chaque méthode associée à l'envoi d'un message décrit dans un diagramme de séquence et puis faire l'union de l'ensemble des permissions. Nous pouvons ainsi gérer la relation F-P de manière automatique [GH00b].

3.4.2.1 Les règles du processus de création des rôles

Nous rappelons que le processus de réalisation du contrôle d'accès d'un système d'information est découpé en deux étapes :
- niveau concepteur de l'application dans lequel on détermine les rôles du système d'information,
- niveau administrateur de sécurité qui définit les associations entre les rôles et les utilisateurs du système.

Un utilisateur est associé à un profil d'utilisateur. Un profil d'utilisateur est défini par l'ensemble des rôles qu'il peut jouer.
Un profil d'utilisateur est défini par un couple $\langle u, listeRoles \rangle$ **:**

1. u est un utilisateur,
2. $ensembleRoles$ est un ensemble de rôles associés à un utilisateur.

On peut donner les règles pour la création des rôles et leurs attachements aux utilisateurs :

1. Il faut définir un profil pour chaque utilisateur qui peut interagir avec le système

$$profil\ d'utilisateur\ attaché\ à\ l'utilisateur$$

$$u_i \vdash profil\ d'utilisateur_i$$

2. Un profil est défini par un ensemble de rôles qu'un utilisateur peut prendre

$$ensemble\ de\ roles\ attachés\ au\ profil\ d'utilisateur$$

$$profil\ d'utilisateur_i \vdash ensembleRoles_i$$

pour obtenir un profil significatif, tout utilisateur devra être au moins attaché à un rôle

3. Un rôle est défini par un ensemble de fonctions

$$ensemble\ de\ fonctions\ attachées\ au\ role$$

$$r_j \vdash ensembleFonctions_j$$

pour obtenir un rôle significatif, tout rôle devra être au moins attaché à une fonction

Au niveau de la description UML, en regardant le diagramme de cas d'utilisation tout acteur doit être attaché au moins à un cas d'utilisation

$$ensemble\ de\ cas\ d'utilisation\ attachés\ à\ l'acteur$$

$$a_j \vdash ensembleCasUtilisation_{a_j}$$

4. Une fonction est définie par un ensemble de permissions nécessaires à l'exercice de celle-ci

$$ensemble\ de\ permissions\ attachées\ à\ la\ fonction$$

$$f_k \vdash ensemblePermissions_k$$

Pour obtenir une fonction significative, une fonction devra être au moins attachée à une permission.

Au niveau de la description UML, en utilisant le diagramme d'interaction (diagramme de séquence ou diagramme de collaboration) chaque cas d'utilisation doit être défini par une description détaillée, i.e. représenté par un ensemble de méthodes exécutées sur les objets

$$ensemble\ de\ couples\ (méthode,\ objet)\ attachées\ au\ cas\ d'utilisation$$

$$uc_k \vdash ensemblePrivileges_{uc_k}$$

3.4.2.2 La création des rôles

Cette section présente comment obtenir un ensemble de rôles du système d'information à partir des différents diagrammes d'UML :

- associer un ***ensemble de privilèges*** à un ***cas d'utilisation*** permettant de définir une fonction,
- associer un ensemble de ***cas d'utilisation*** (fonctions) à un ***acteur*** permettant de définir un rôle.

Détermination des permissions associées à une fonction

Un cas d'utilisation est la description des besoins d'un acteur. Il permet de préciser les limites des interactions de celui-ci. Les interactions entre les objets sont représentées par les séquences d'actions, ce qui permettra de définir un ensemble de privilèges.

Donc, on part d'un diagramme de séquence qui représente un scénario possible associé à une fonction particulière (i.e. cas d'utilisation). Le parcours du diagramme de séquence permet de trouver les permissions associées à une fonction.

Chaque message $msg(o_1, o_2, m)$, dans une interaction, envoyé d'un objet o_1 à un objet o_2 pour exécuter une méthode m sur un objet o_2 doit être attaché à une permission correspondant à la possibilité d'exécuter la méthode m sur l'objet o_2. Quand on prend les contraintes (Chapitre 2 et Chapitre 4) en considération, une définition d'un message est donnée comme suit : $msg(o_1, o_2, m, cst)$ ou cst est la contrainte [GH00b].

Un ensemble de permissions pour une interaction i est défini :

$$P(i) = \{p | \varphi(msg(o_1, o_2, m, cst)) = p(m, o_2) \wedge cst(p) = vrai\}$$

où φ est une fonction qui attache une permission au message msg et o_1 est un acteur ou une instance d'une classe qui peut exécuter une méthode m sur un objet o_2.

Un diagramme d'interaction, en particulier un diagramme de séquence est défini par un ensemble d'interactions, alors un ensemble de permissions doit être attaché à ce diagramme d. Il est décrit par un ensemble D d'interaction

$$P(d) = \bigcup_{i \in D} P(i)$$

Un cas d'utilisation μ est décrit par un ensemble M de diagrammes d'interactions d_i, on peut alors déduire les permissions qui seront attachées à celui-ci par :

$$P(\mu) = \bigcup_{d_i \in M} P(d_i)$$

L'ensemble $P(\mu)$ représente un ensemble de permissions attachées à une fonction particulière représentée par un cas d'utilisation μ.

Détermination des fonctions associées à un rôle

Le diagramme de cas d'utilisation permet de visualiser un ensemble de fonctionnalités d'un système en examinant les besoins de chaque acteur. Ceux-ci sont exprimés sous forme de cas d'utilisation.

Donc, on part d'un diagramme de cas d'utilisation qui représente les relations entre les acteurs et les cas d'utilisation. Le parcours de ce diagramme donne un ensemble de fonctions qui doivent être attachées à un rôle particulier.

Un diagramme de cas d'utilisation dcu_i contient des cas d'utilisation (i.e. fonctions) associés à plusieurs acteurs (i.e. rôles). L'ensemble des fonctions attachées à un rôle r particulier est défini par les fonctions qui sont en relation directe avec le rôle ou en relation indirecte par les relations d'héritage entre les fonctions :

$$F(r_{dcu_i}) = \{f \mid f = uc\, pour\, uc \in dcu_i \wedge (r_{dcu_i}, f) \in R - F\} \cup$$

$$\cup \{f' \mid f' = uc\, pour\, uc \in dcu_i \wedge ((f, f') \in F - F \wedge (r_{dcu_i}, f) \in R - F)\}$$

Par exemple, en utilisant le diagramme de cas d'utilisation présenté dans la Figure 3.11, les privilèges de *Validation d'utilisateur* seront ajoutés à l'ensemble des privilèges de *Configuration*.

En considérons le diagramme de cas d'utilisation présenté dans la Figure 3.20, l'ajout de l'ensemble des privilèges du cas d'utilisation *visualiser* à l'ensemble des privilèges du cas d'utilisation *visualiser le bulletin* permettrait à *Etudiant* d'avoir la possibilité de *visualiser la liste complète* qui serait non conforme à l'utilisation de <<extends>>. Par contre, l'ajout de l'ensemble des privilèges de *visualiser le bulletin*, de *visualiser la liste complète* et de *visualiser les notes* à l'ensemble des privilèges de *visualiser* permettrait aux acteurs *Directeur des Etudes*, *Enseignant* et *Secrétariat* de faire une visualisation selon les trois types et à *Etudiant* - selon les deux types, ce qui serait convenable. Pour résoudre ce problème, il faut ajouter les fonctions *visualiser le bulletin*, *visualiser la liste complète* et/ou *visualiser les notes* aux ensembles de fonctions associées aux acteurs *Directeur des Etudes*, *Enseignant* et *Secrétariat*.

L'ensemble des fonctions de ce rôle est défini par l'union des cas d'utilisation associés à ce rôle dans tous les diagrammes de cas d'utilisation D_{cu} :

$$F(r) = \bigcup_{dcu_i \in D_{cu}} F(r_{dcu_i})$$

Pour construire l'ensemble des rôles associés au profil d'un utilisateur il faut réaliser l'affectation des rôles aux utilisateurs. Cette étape sera sous la responsabilité de l'administrateur de sécurité.

Pour résumer notre proposition on peut dire que :
- un privilège (i.e. autorisation) représente le droit d'exécution d'une méthode sur un objet avec vérification de contraintes,
- un ensemble de privilèges est associé à un cas d'utilisation qui représente une fonction,
- un ensemble de fonctions est associé à un acteur qui représente un rôle,
- un ensemble de rôles détermine un profil d'utilisateur.

L'algorithme de création des rôles (i.e. le processus de détermination des ensembles $P(\mu)$ et $F(r)$) en utilisant le méta-modèle UML est présentée dans le Chapitre 5.

3.5 Conclusion

La complexité des systèmes d'information provoque le développement de nouvelles démarches et de nouveaux outils. Ainsi, ces dernières années l'approche objet a pris de l'extension et les principaux courants ont fusionné pour donner le langage UML.

Nous avons donné dans ce chapitre une proposition d'extension du modèle RBAC classique permettant plus de flexibilité au niveau de la gestion des rôles pour mieux s'adapter aux organisations complexes et évolutives des entreprises actuelles.

En se basant sur le rapprochement des concepts du modèle RBAC étendu aux concepts d'UML, on a montré la possibilité d'intégrer un contrôle d'accès de type RBAC au niveau de la conception du système d'information.

Le langage UML nous permet d'implémenter le modèle RBAC étendu en définissant les droits d'accès aux objets du système d'information en accord avec le modèle de sécurité choisi.

Le chapitre suivant introduit la notion de contrainte au niveau du contrôle d'accès d'un système d'information. Les contraintes sont classées en deux catégories :
- les contraintes d'application spécifiés par le concepteur de l'application,
- les contraintes liées à l'organisation de l'entreprise qui seront spécifiées par l'administrateur de sécurité.

Chapitre 4

Contraintes et Cohérence

Objectifs du chapitre :
- *Définir la notion de contrainte de sécurité*
- *Introduire le langage OCL et le langage RCL 2000*
- *Présenter une classification des contraintes*
- *Examiner les contraintes de différents points de vues*
- *Décrire la confrontation des points de vues*
- *Développer la chaîne d'intégration pour garantir la cohérence globale du système d'information*

Le processus d'administration de la sécurité dans un système d'information est une tâche complexe. Beaucoup de contraintes de sécurité doivent être exprimées dans le but de bien définir la politique de sécurité.

Une ***contrainte de sécurité*** est une information associée aux éléments du système qui spécifie les conditions à satisfaire pour garantir les règles de sécurité et la cohérence globale du système d'information.

Les contraintes de sécurité peuvent être classifiées dans deux groupes. Les contraintes du premier groupe sont celles exigées par l'application. Elles sont validées au niveau de la conception de celle-ci par le concepteur de l'application. Les contraintes du deuxième groupe concernent la politique de sécurité globale du système d'information. Elles sont proposées au niveau de l'organisation par un administrateur global de sécurité [CS95, San90, San96].

Le premier groupe contient toutes les contraintes de sécurité spécifiques à une application. Elles donnent la possibilité de gérer une application complexe. Les développeurs d'application connaissent bien leur application, ils peuvent décrire facilement les contraintes de sécurité de l'application. L'administrateur de sécurité, qui connaît la politique globale mise en place dans le système d'information, est la personne la plus adaptée pour définir les contraintes au niveau global. La principale difficulté est de garder la cohérence de l'ensemble lors de l'ajout d'une application avec l'introduction de nouveaux rôles ou de nouvelle permissions.

Les contraintes de ces deux groupes doivent être exprimées en utilisant des outils standards. Nous utilisons pour cela deux langages pour exprimer les contraintes - le langage OCL [Gro00a, WK99b] et le langage RCL 2000 [Ahn99].

Ce chapitre présente les contraintes du point de vue du concepteur et du point de vue de l'administrateur de sécurité. On présente les différents types de contrainte et leurs représentations en utilisant deux outils - le langage OCL et le langage RCL 2000. La troisième partie du chapitre décrit la validation de deux groupes de contraintes (Figure 4.1) pour garantir la cohérence globale du système d'information.

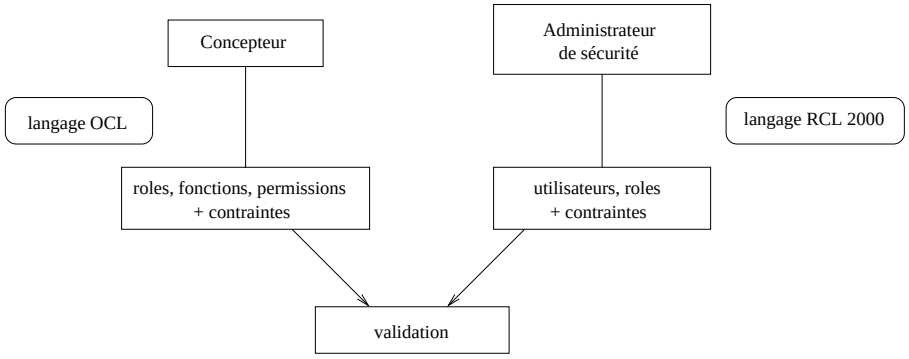


FIG. 4.1 – Le concepteur et l'administrateur d'une application.

4.1 Les langages de contraintes

Les deux langages utilisés pour la représentation des contraintes sont : le langage OCL [WK99a, WK99b] et le langage RCL 2000 [Ahn99, AS99]. Le langage OCL permet d'exprimer des contraintes dans le langage UML. Le langage RCL 2000 a été créé pour la présentation des contraintes dans le modèle RBAC.

4.1.1 Le langage OCL

Une ***contrainte*** est une information sémantique associée aux éléments d'un modèle qui spécifie les conditions que le modèle doit satisfaire pour être correct.

UML définit un langage d'expression de contrainte appelé **OCL (Object Constraint Language)** [Gro00a].

Le résultat d'une expression OCL est une valeur qui n'a aucun effet sur l'application ; l'évaluation d'une expression ne change rien à une application. OCL n'est pas un langage de programmation. OCL est un langage typé dans le sens où chaque expression possède un type.

OCL est utile pour les spécifications et descriptions suivantes :
- spécifier des invariants sur les classes dans le modèle de classes,
- spécifier des invariants pour les stéréotypes,
- décrire des pré- et post-conditions sur les opérations,
- décrire des conditions de garde.

4.1.1.1 Les types de contraintes en OCL

On peut définir trois types de contraintes en utilisant le langage OCL [RG, SS, Ken97] :
- ***pré-condition*** - est un prédicat qui doit être vérifié avant l'exécution de l'opération ou de la méthode,
- ***post-condition*** - est un prédicat qui doit être vérifié après l'exécution de l'opération ou de la méthode,
- ***invariant*** - définit une condition qui doit être toujours vérifiée pour chaque instance de la classe, type ou interface.

Les contraintes sous forme d'invariant sont attachées aux classes et aux propriétés auxquelles ces contraintes se rapportent. Les contraintes sous formes de pré- ou post-conditions sont attachées aux méthodes (messages) dans un diagramme de séquence.

4.1.1.2 Les concepts d'OCL pour la présentation des contraintes

Une expression d'OCL peut être utilisée dans chaque diagramme d'UML pour spécifier une condition sur un objet ou un ensemble d'objets. On peut utiliser une expression d'OCL :

- dans un diagramme de classe, sur les attributs et les opérations de classe,
- dans un diagramme de séquence ou un diagramme de collaboration, un message peut avoir une condition attachée, qui spécifie dans quelle circonstance ce message est envoyé,
- dans un diagramme de séquence ou dans un diagramme de collaboration, des messages peuvent avoir des paramètres ; les valeurs de ces paramètres peuvent être spécifiées en utilisant une expression d'OCL,
- un message est envoyé à un objet précis spécifié par une expression d'OCL.

Il y a différentes façons d'exprimer la même information. Les contraintes peuvent être spécifiées dans différents diagrammes d'UML. Le choix du diagramme et de la présentation retenus sont importants pour la clarté et la lisibilité d'un modèle. Nous avons choisi de présenter les contraintes dans les diagrammes de classes ou dans les diagrammes de séquence.

La contrainte d'invariant
On peut distinguer dans l'expression d'une contrainte deux parties :

context Classe inv :
Condition

1. le nom de la classe - *Classe* dans laquelle la contrainte est définie,
2. la condition de la contrainte - *Condition* est une condition (ou un ensemble de conditions) - une expression booléenne qui doit être vérifiée pour tous les éléments de la classe.

La contrainte de pré-condition
On peut distinguer dans l'expression d'une contrainte deux parties :

context Classe : : Methode
pre : objet | objetCondition.

1. le nom de la classe - *Classe* et le nom de la méthode - *Methode* dans laquelle la contrainte est définie,
2. la condition de la contrainte :
   (a) la contrainte est définie comme une pré-condition - le mot *"pre"*,
   (b) l'objet *objet* de la classe *Classe*,
   (c) la condition à vérifier pour les objets de la classe - *objetCondition*.

La contrainte de post-condition
La forme de présentation de la post-condition est identique à la pré-condition :

context Classe : : Methode : typeRetourne
post : result = objet | objetCondition

On a ajouté dans cette définition *typeRetourne* qui donne le type retourné par la méthode *Methode*. On utilise le mot *"post"* pour indiquer une post-condition et le mot *"result"* pour retourner le résultat de la méthode. Celui-ci doit être du même type que le type donné par *typeRetourne*.

Les contraintes dans les diagrammes de classes sont souvent présentées comme des invariants, par exemple :

context Etudiant inv :
$self.age \leq 30$

limite l'age des étudiants à 30 ans.
Les contraintes dans les diagrammes de séquence peuvent être exprimées comme des pré- ou post-conditions, par exemple

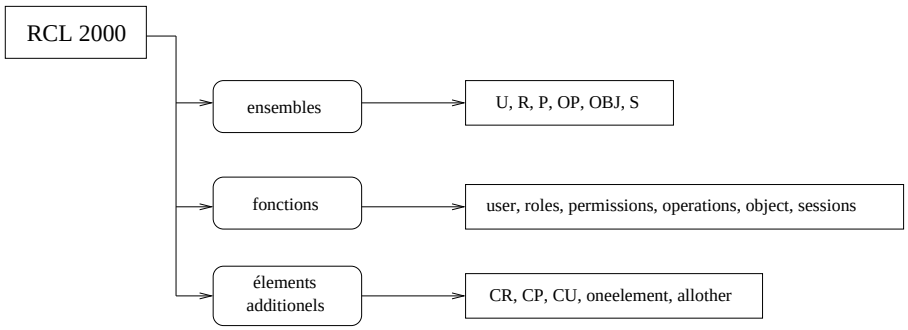


FIG. 4.2 – Les éléments du langage RCL 2000.

la pré-condition :

$context\ Note\ ::\ getValeur()\ :\ double$
$pre\ :\ note \mid note.etudiant.getNom() = PEtudiant.getName()$

limite l'accès des étudiants à leurs propres notes,
la post-condition :

$context\ Note\ ::\ setValeur()\ :\ void$
$post\ :\ result = note \mid note \in \{1, 2, ..., 20\}$

limite l'intervalle des valeurs de notes données par un enseignant.

4.1.2 Le langage RCL 2000

Le langage RCL 2000 a été créé pour spécifier des contraintes dans un modèle RBAC. L'utilisateur de ce langage peut être un administrateur de sécurité qui doit réaliser des contraintes d'administration basées sur les rôles [Ahn99].

4.1.2.1 La présentation du langage RCL 2000

Le langage **RCL 2000 (Role-based Constraints Language 2000)** est un langage de spécification de contraintes basées sur les rôles [Ahn99, AS99]. Les concepts manipulés dans le langage RCL 2000 sont : les ensembles, les fonctions et les éléments additionnels comme CR, CP, CU définis en accord avec le modèle RBAC (Figure 4.2).

Le langage RCL 2000 contient six ensembles d'entités : *users (U), roles (R), objects (OBJ), operations (OP), permissions (P)* et *sessions (S)*. Ce sont les ensembles définis dans un modèle RBAC. On peut les présenter comme suit [Ahn99] :

- un ensemble d'utilisateurs : $U = \{u_1, u_2, ..., u_n\}$,
- un ensemble de rôles : $R = \{r_1, r_2, ..., r_m\}$,
- un ensemble d'objets : $OBJ = \{obj_1, obj_2, ..., obj_p\}$,
- un ensemble d'opérations : $OP = \{op_1, op_2, ..., op_o\}$,
- un ensemble de permissions : $P = \{p_1, p_2, ..., p_s\}$, où $P = OP \times OBJ$,
- un ensemble de sessions : $S = \{s_1, s_2, ..., s_r\}$.

Un ensemble de fonctions a été défini pour manipuler les ensembles introduits précédemment (Figure 4.2). Des éléments additionnels permettent de représenter la complexité des fonctions du système. Certains éléments de ces ensembles peuvent être conflictuels, c'est-à-dire qu'il existe une contrainte qui interdit leurs utilisations simultanées. Des rôles qui s'excluent mutuellement ne peuvent pas être utilisés en même temps par le même individu. Ces rôles sont définis comme des *rôles en conflit* - des *rôles conflictuels*. L'ensemble **CR** contient les ensembles de rôles qui sont définis comme rôles en conflit. De la même façon, l'ensemble **CP** se compose d'une collection de *permissions en conflit* et **CU** d'une collection d'*utilisateurs en conflit*.

Ces ensembles peuvent être présentés de la manière suivante :
- une collection d'ensembles de rôles en conflit : $CR = \{cr_1, cr_2, ..., cr_n\}$ où $cr_i = \{r_i, .., r_t\} \subseteq R$,
- une collection d'ensembles de permissions en conflit : $CP = \{cp_1, cp_2, ..., cp_m\}$ où $cp_i = \{p_i, .., p_s\} \subseteq P$,
- une collection d'ensembles d'utilisateurs en conflit : $CU = \{cu_1, cu_2, ..., cu_p\}$ où $cu_i = \{u_i, .., u_x\} \subseteq U$.

Pour faciliter les démonstrations de cohérence des règles de sécurité, deux quantificateurs ont été introduits :
- $oneelement\,(X) = x_i$ où $x_i \in X$, donne la possibilité de prendre un élément quelconque x_i de l'ensemble X ; on peut écrire cette fonction *oneelement* comme OE ; chaque occurrence de $OE\,(X)$ dans la même expression se réfère à chaque fois au même élément x_i de l'ensemble X,
- $allother\,(X) = X - \{OE\,(X)\}$ donne l'ensemble des éléments de l'ensemble X auquel on a retiré l'élément identifié par l'opérateur $OE\,(X)$; on peut écrire *allother* comme AO.

Les fonctions définies dans le langage RCL 2000 ne contiennent pas de variables de temps ou d'état. Chaque fonction considère le temps courant ou l'état courant. En général pour une fonction f sur un ensemble X :

$$f\,(X) = f\,(x_1) \cup f\,(x_2) \cup ... \cup f\,(x_n)$$

où $X = \{x_1, x_2, ..., x_n\}$.

Par exemple, si on veut définir les utilisateurs qui sont attachés à un ensemble de rôles $R = \{etudiant1, etudiant2, etudiant3\}$, on utilise la fonction $user\,(R)$ et on obtient ce qui suit :

$$user\,(R) = user\,(etudiant1) \cup user\,(etudiant2) \cup user\,(etudiant3)$$

On a présenté le langage RCL 2000 et ses composants de base. Ils sont basés sur le modèle RBAC. La syntaxe formelle et la sémantique sont présentées en détails dans [Ahn99, AS99].

4.1.2.2 L'extension du langage RCL 2000

Notre but est d'utiliser le formalisme du langage RCL 2000 pour prouver la cohérence des règles de sécurité d'un modèle RBAC étendu (présenté dans le Chapitre 3). Pour cela nous avons introduit quelques nouveaux éléments dans le langage RCL 2000 pour qu'il puisse tenir compte des extensions du modèle RBAC.

De nouveaux ensembles ont été ajoutés aux six ensembles d'entités définis en RCL 2000 : un ensemble de *fonctions (F)* : $F = \{f_1, f_2, ..., f_t\}$, un ensemble de *méthodes (M)* : $M = \{m_1, m_2, ..., m_w\}$ et un ensemble de *classes (C)* : $C = \{c_1, c_2, ..., c_z\}$. Ils ont été introduits pour tenir compte du modèle objet utilisé.

Nous allons utiliser dans ce chapitre le symbole O pour indiquer un ensemble d'objets, $O = \{o_1, o_2, ..., o_p\}$.

Les relations qui existent entre les éléments du modèle RBAC étendu sont représentées comme suit :
- $RH \subseteq R \times R$ - représente une hiérarchie de rôles,
- $FH \subseteq F \times F$ - représente une hiérarchie de fonctions,
- $U - R \subseteq U \times R$ - une relation d'affectation d'un utilisateur aux rôles,
- $R - F \subseteq F \times R$ - une relation d'affectation des fonctions aux rôles,
- $F - P \subseteq P \times F$ - une relation d'affectation des permissions aux fonctions.

Nous avons aussi modifié l'ensemble des fonctions définies en RCL 2000 :
- déjà définie : $user : S \rightarrow U$ donne l'utilisateur attaché à une session s_i,

- déjà définie : $user : R \rightarrow 2^U$ donne pour chaque rôle r_i un ensemble d'utilisateurs attachés à ce rôle, où $user\,(r_i) = \{u \in U \mid (u, r_i) \in U - R\}$,
- $roles : U \cup F \cup S \rightarrow 2^R$ donne un ensemble de rôles attachés à un utilisateur u_i ou à une fonction f_i, ou donne un ensemble de rôles actifs dans une session s_i, où
  $roles\,(u_i) = \{r \in R \mid (u_i, r) \in U - R\}$,
  $roles\,(f_i) = \{r \in R \mid (r, f_i) \in R - F\}$,
- $roles^\star : U \cup F \cup S \rightarrow 2^R$ étend la fonction $roles$ avec la notion de hiérarchie de rôles,
- $sessions : U \cup R \rightarrow 2^S$ donne l'ensemble des sessions ouvertes par un utilisateur u_i ou donne l'ensemble des sessions qui ont activé un rôle r_i,
- $functions : R \cup P \rightarrow 2^F$ donne un ensemble de fonctions attachées à un rôle r_i ou à une permission p_i, où
  $functions\,(r_i) = \{f \in F \mid (r_i, f) \in R - F\}$,
  $functions\,(p_i) = \{f \in F \mid (f, p_i) \in F - P\}$,
- $functions^\star : R \cup P \rightarrow 2^F$ étend la fonction $functions$ avec la notion de hiérarchie de fonctions,
- $permissions : F \rightarrow 2^P$ donne pour chaque fonction f_i un ensemble de permissions affectées à cette fonction, où $permissions\,(f_i) = \{p \in P \mid (f_i, p) \in F - P\}$,
- $methods : F \times O \rightarrow 2^M$ affecte à chaque couple fonction f_i et objet o_i un ensemble de méthodes qui peuvent être réalisées sur cet objet,
- déjà définie : $object : P \rightarrow 2^O$ donne pour chaque permission p_i un ensemble d'objets sur lesquels cette permission est définie,
- $operation : M \rightarrow OP$ donne pour chaque méthode m_i l'opération de cette méthode,
- $class : O \rightarrow C$ donne pour chaque objet o_i la classe de cet objet,
- $instances : C \rightarrow 2^O$ donne pour chaque classe c_i un ensemble d'instances de cette classe.

On peut aussi définir, de la même façon que les ensembles CU, CR et CP, un ensemble de fonctions en conflit :

une collection d'ensembles de fonctions en conflit : $CF = \{cf_1, cf_2, ..., cf_o\}$ où $cf_i = \{f_1, ..., f_w\} \subseteq F$.

4.2 Les contraintes

4.2.1 Classification des contraintes

Les contraintes qui sont utilisées dans l'organisation de la sécurité d'un système d'information peuvent être classifiées en fonction de différents critères. Nous proposons une classification qui s'inscrit dans le cycle de vie d'une application : analyse - développement et exploitation - maintenance. Dans la phase d'*analyse - développement*, le concepteur de l'application impose des contraintes d'utilisation qui garantiront à terme la cohérence des données qui seront manipulées par l'application. Dans la phase d'*exploitation - maintenance* l'application est insérée dans un système d'information existant. Cette insertion doit respecter la cohérence globale du système par le biais de contraintes, dites globales.

Ceci nous donne la classification de contraintes suivante :

- contraintes du point de vue du concepteur - contraintes définies au niveau de l'application d'un système,
- contraintes du point de vue de l'administrateur - contraintes définies au niveau de la sécurité globale d'une entreprise.

Nous pouvons ensuite donner les contraintes associées au modèle RBAC :

- *contraintes sur une permission* - ces contraintes limitent l'ensemble des objets accessibles par une permission,
- *contraintes de séparation des responsabilités* (contraintes SOD) - ces contraintes représentent le concept de rôles en exclusion mutuelle et de permissions en exclusion mutuelle,

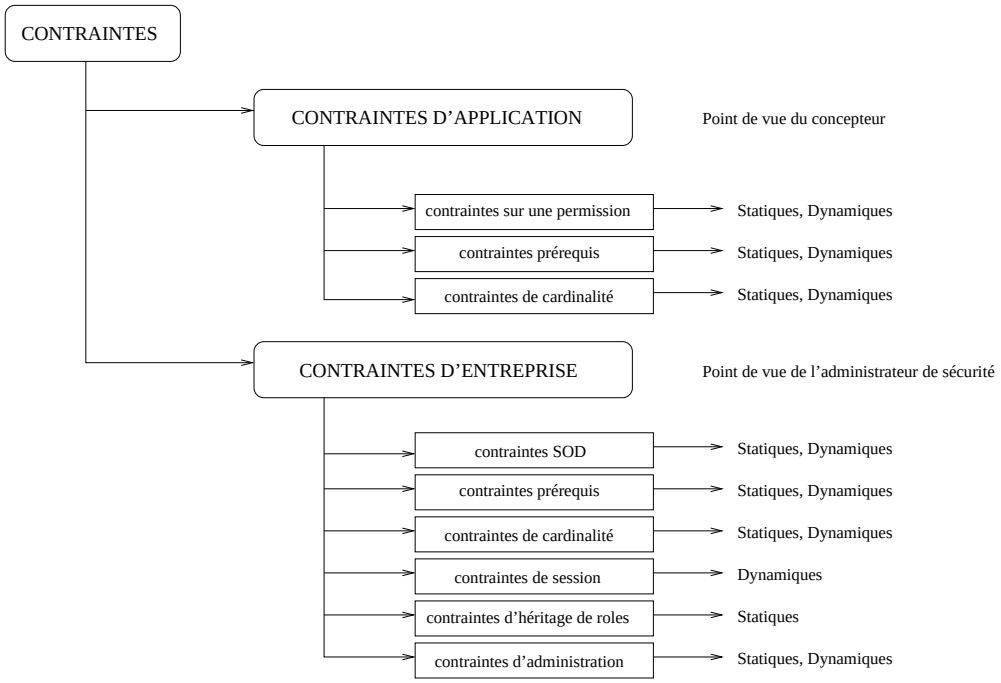


FIG. 4.3 – La classification des contraintes.

- *contraintes pré-requis* - ces contraintes sont basées sur le concept de rôle pré-requis, par exemple un utilisateur peut être affecté au rôle r1 seulement si cet utilisateur est déjà affecté au rôle r2,
- *contraintes de cardinalité* - ce type de contrainte est la limitation numérique des éléments d'un ensemble du modèle RBAC étendu,
- *contraintes sur une session* et *contraintes sur une hiérarchie de rôles* du modèle RBAC,
- *contraintes d'administration* - ces contraintes sont associées à un modèle ARBAC.

Certaines contraintes doivent être satisfaites indépendamment de l'utilisateur qui va les tester. Par exemple lorsqu'il s'agit d'une contrainte sur un domaine de valeurs que peut prendre une donnée. Ces contraintes sont dites *statiques*. Elles existent indépendamment de l'état du système. D'autres, par contre, sont associées à l'identification de l'utilisateur qui exécute une action. Elles sont dites *dynamiques*. Elles apparaissent au cours d'un changement d'état du système.

La Figure 4.3 présente la classification des contraintes selon les trois niveaux précédents. En accord avec cette classification, nous présentons ci-dessous les contraintes du point de vue du concepteur et du point de vue de l'administrateur de sécurité. Nous utilisons deux langages pour la représentation des contraintes : le langage OCL au niveau du concepteur et le langage RCL 2000 au niveau de l'administrateur.

4.2.2 Le point de vue du concepteur

Cette section indique les contraintes qu'un concepteur peut ou veut exprimer. Le langage OCL a été créé pour permettre l'ajout de contraintes dans les différents diagrammes UML. Il est donc naturel que le concepteur utilise ce langage pour exprimer ses contraintes de sécurité.

Au niveau de la conception, le langage UML présente l'utilisateur comme un acteur qui joue un rôle vis à vis du système. Dans la phase d'exploitation, l'administrateur de sécurité utilise pour valider les règles d'accès le concept d'utilisateur. Pour favoriser la communication du schéma de sécurité proposé par le concepteur à l'administrateur nous avons utilisé un acteur que l'on a

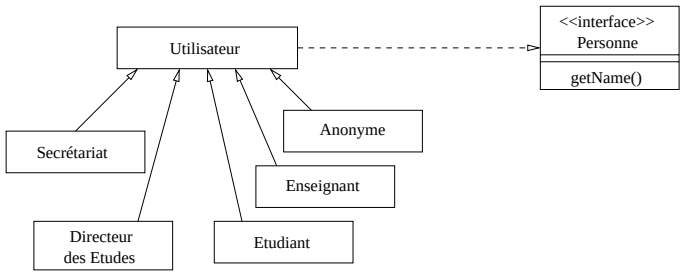


FIG. 4.4 – Le concept d'un utilisateur dans le modèle de l'application "Gestion des Notes".

nommé *Utilisateur*.

Un utilisateur peut être donné sous la forme d'une classe dans le diagramme de classe qui implémente l'interface "*Personne*". Cette interface peut contenir différents attributs et méthodes (par exemple la méthode *getName()*). Elle peut être implémentée par les utilisateurs de ce système. L'identification d'un utilisateur dans un système est obtenue par la méthode *getName()* - on exécute pour cela *Personne.getName()*. Le concept d'utilisateur dans l'exemple d'application "Gestion des Notes" peut être représenté comme dans la Figure 4.4.

Nous proposons aussi définir la classe *Permission* qui représente une permission d'exécuter une méthode sur un objet dans un diagramme de séquence décrivant un cas d'utilisation.

4.2.2.1 Les types de contraintes manipulées par le concepteur

On distingue trois types de contrainte au niveaux de la conception : les contraintes sur une permission, les contraintes de pré-requis et les contraintes de cardinalité.

Au niveau de la conception, on gère uniquement des contraintes sur les permissions. L'étape de conception se base sur un cahier des charges qui décrit seulement des autorisations positives avec l'hypothèse d'un monde fermé.

Contraintes sur une permission

Il y a deux types de limitation d'accès sur une permission :

- ***statique*** - les contraintes statiques réduisent l'ensemble des objets accessibles par une méthode indépendamment de l'utilisateur qui exécutera la méthode,
- ***dynamique*** - contrainte en relation avec l'identité de la personne qui utilise la permission ; les contraintes dynamiques réduisent l'ensemble des objets accessibles par une méthode au cours d'une session.

Une permission peut être définie comme une fonction : $p(m, o)$ qui retourne le résultat *vrai* ou *faux* : ***vrai*** - quand l'autorisation est donnée et ***faux*** - quand l'autorisation n'est pas donnée.

Une contrainte sur une permission caractérise l'ensemble des objets sur lesquels on pourra exécuter la méthode m :

$$cst : P \rightarrow 2^O \text{ et } cst(p) = \{o \mid o \in O_C \subset O \wedge o\ v\acute{e}rifie\ un\ pr\acute{e}dicat\ \pi \wedge p(m, o) = vrai\}$$

où $m \in Methodes(C)$, $Methodes(C) - l'ensemble\ des\ m\acute{e}thodes\ de\ la\ classe\ C$

$$O_C - l'ensemble\ des\ objets\ instances\ de\ la\ classe\ C$$

π est un prédicat qui doit être vérifié pour chaque objet de la classe C.

On présente deux exemples de contrainte, une statique et une dynamique qui sont exprimées en utilisant le langage OCL :

1. contrainte statique :
   par exemple : la permission d'écrire dans les documents ayant l'extension "doc"

$context\ Fichier\ :\ ecrire()$
$pre\ :\ Set\,(Objects) \rightarrow select\,(obj \mid obj.oclIsTypeOf(Fichier)\ and\ obj.getName() =' *.doc')$
où $obj.getName()$ est le nom de l'objet obj

2. contrainte dynamique :
   par exemple : la permission d'écrire dans les documents dont on est le propriétaire des fichiers :
   $context\ Fichier\ :\ ecrire()$
   $pre\ :\ Set\,(Objects) \rightarrow$

$$select\,(obj \mid obj.oclIsTypeOf(Fichier)\ and\ obj.proprietaire = Personne.getName())$$

   où $obj.proprietaire$ est le propriétaire de l'objet obj

Contraintes de pré-requis et contraintes de cardinalité

Une contrainte de pré-requis précise qu'une fonction p peut être attachée à un rôle si et seulement si le rôle possède déjà une fonction q. Par exemple : dans un système de fichiers, la permission de lire un fichier nécessite la permission de lire le répertoire dans lequel ce fichier est situé (pour pouvoir lister les fichiers d'un répertoire, il faut avoir le droit de lecture sur ce répertoire)

$context\ SystemeFichiers\ inv$:

$$self.permission \rightarrow includes\,(p\,('lire',\ 'fichier'))\ \ implies$$

$$implies\ self.permission \rightarrow includes\,(p\,('lire',\ 'fichier.repertoire'))$$

Les contraintes de cardinalité sont des limitations numériques sur des éléments d'une application. Par exemple : la permission de lire le fichier des mots de passe du système peut être associée seulement à un rôle spécifique

$context\ Permission\ inv$:

$$(self.methode \rightarrow includes\,('lire')\ and\ self.objet \rightarrow includes\,('MotsDePasse')) \rightarrow$$

$$implies\ self.cas\ d'utilisation.acteur = 1$$

4.2.3 Le point de vue de l'administrateur

Cette section indique les contraintes qu'un administrateur de sécurité peut ou veut exprimer. L'administrateur de sécurité doit garder la cohérence globale du système d'information en accord avec les règles de sécurité définies par l'entreprise. Des contraintes exprimées au niveau de l'administrateur doivent implémenter ces règles.

Les contraintes présentées du point de vue de l'administrateur sont des contraintes définies en particulier sur un ensemble d'utilisateurs et sur un ensemble de rôles. Ces types de contrainte sont présentés dans [SCFY96, AS99].

Le langage RCL 2000 a été défini pour exprimer de manière formelle les contraintes dans le modèle RBAC. C'est pourquoi on a choisi ce langage dans la suite de nos travaux.

4.2.3.1 Les types de contraintes manipulées par l'administrateur de sécurité

On peut citer quelques types de contrainte du point de vue de l'administrateur de sécurité.

Contraintes SOD

Les *contraintes de séparation des responsabilités* sont une classe de contraintes importante pour prévenir des erreurs dans les politiques de sécurité des systèmes. Par exemple, on peut définir des rôles qui s'excluent mutuellement. Un exemple classique de rôles qui s'excluent mutuellement est : le rôle d'*agent de vente* et le rôle d'*agent de paiement*. En général, pour éviter toute opération frauduleuse, un même individu ne peut pas être attaché à ces deux rôles en même temps.

On peut donner une définition de SOD [Ahn99] :

*Une **séparation des responsabilités** réduit par le partage de tâches et des privilèges associés, la possibilité de tricheries ou d'erreurs importantes qui peuvent causer un dommage dans une organisation.*

On peut spécifier les différents types de contraintes SOD en fonction des concepts sur lesquels s'applique la contrainte : les *contraintes sur les rôles en conflit,* les *contraintes sur les utilisateurs en conflit* et les *contraintes sur les permissions en conflit.*

On a de nouveau le découpage statique ou dynamique des contraintes SOD :

- *SOD statiques* - les plus simples, contiennent les contraintes qui sont définies statiquement, avant la réalisation par un utilisateur des différentes activités dans un système.
- *SOD dynamiques* - contraintes que respectent les rôles activés par un utilisateur dans une session. Elles sont basées sur les actions qui ne peuvent être réalisées simultanément.

Ces types de contrainte sont présentées dans [Ahn99, AS99, San90, CS95] en utilisant le langage RCL 2000.

Contraintes de pré-requis

L'administrateur de sécurité peut aussi définir les *contraintes de pré-requis* - ces contraintes sont basées sur le concept de rôle pré-requis ; par exemple : un utilisateur peut être affecté à un rôle $r1$ si et seulement si il est déjà affecté à un rôle $r2$.

$$r1 \in roles\,(OE\,(U)) \;\Rightarrow\; r2 \in roles\,(OE\,(U))$$

Par exemple un utilisateur peut être affecté au rôle *Enseignant* seulement si cet utilisateur est déjà affecté au rôle *Employé* ; cela garantit que seuls les utilisateurs qui sont déjà affectés au rôle *Employé* peuvent être affectés au rôle *Enseignant*

$$'Enseignant' \in roles\,(OE\,(U)) \;\Rightarrow\; 'Employé' \in roles\,(OE\,(U))$$

Contraintes de cardinalité

Un autre type de contrainte au niveau de l'administrateur est la *contrainte de cardinalité* :

- le nombre des utilisateurs affectés à un rôle peut être limité ; par exemple, le rôle *Directeur des Etudes* doit être affecté seulement à un seul utilisateur

$$|user\,('Directeur\ des\ Etudes')| = 1$$

- le nombre des sessions qu'un utilisateur peut activer dans le même temps est limité ; par exemple, un utilisateur peut ouvrir seulement trois sessions simultanément

$$|sessions\,(OE\,(U))| = 3$$

Contraintes de session et contraintes de hiérarchie de rôles

L'administrateur de sécurité peut aussi définir :

- des *contraintes sur une session*
  - un utilisateur peut être membre de deux rôles $r1,\ r2$ mais il ne pourra pas les activer simultanément dans une même session

$$roles\,(OE\,(U)) = \{r1,\ r2\} \;\wedge\; |sessions\,(r1) \cap sessions\,(r2)| = \emptyset$$

- des *contraintes sur une hiérarchie de rôles*
  - une permission attachée au rôle fils ("junior rôle") ne peut pas être attachée au rôle parent ("senior rôle"),
  - un utilisateur attaché au rôle fils ne peut pas être attaché au rôle parent.

Contraintes d'administration

Nous avons présenté dans le Chapitre 2 le modèle ARBAC - le modèle d'administration pour la gestion du modèle RBAC. On peut aussi citer les contraintes liées avec ce modèle appelées contraintes d'administration. L'utilisation de contraintes administratives a comme objectif la validation de toute modification du modèle de sécurité.

Les rôles administratifs AR et les permissions administratives AP sont disjoints des rôles réguliers R et des permissions régulières P. Ce modèle indique que les permissions peuvent être seulement attachées aux rôles et les permissions administratives peuvent être seulement attachées aux rôles administratifs. Ces règles sont implémentées par un ensemble de contraintes.

Les autorisations d'un administrateur de sécurité lui donnent la possibilité de modifier les relations qui existent entre les éléments du modèle RBAC utilisé. Les permissions qui autorisent ces opérations administratives doivent satisfaire des règles de gestion définies par des contraintes.

Par exemple, on veut ajouter une nouvelle application. Cela peut impliquer l'ajout de nouveaux rôles pour permettre l'utilisation de cette application. L'administrateur de sécurité situe ces rôles dans une hiérarchie de rôles. L'administrateur de sécurité va faire des changements dans la hiérarchie des rôles existante. Il peut ainsi provoquer un conflit ou une incohérence par liaison indirecte de rôles conflictuels. Cela demande la possibilité de contrôler les opérations effectuées par les rôles administratifs en ajoutant de nouvelles contraintes ou en modifiant les ensembles d'éléments en conflit.

Cette situation et les solutions possibles pour les résoudre sont décrites dans [SCFY96, SaQM99]. Le modèle RRA [SaQM99, SM98b] du modèle ARBAC permet de gérer ces solutions.

4.2.4 Les types de contrainte du modèle RBAC étendu

On présente dans cette section les contraintes propres au modèle RBAC étendu que l'on a présenté dans le Chapitre 3. L'introduction de la notion de fonction nécessite de la part du concepteur l'affectation des fonctions aux rôles dans le processus de réalisation du schéma de sécurité. On peut déterminer les contraintes suivantes :

1. contrainte pour une relation **utilisateur - rôle**, par exemple :
   (a) un utilisateur ne peut pas être affecté à deux rôles en conflit
   $$|roles\,(OE\,(U)) \cap OE\,(CR)| \leq 1$$
2. contrainte pour une relation **utilisateur - session**, par exemple :
   (a) un utilisateur possède un nombre limité de sessions ouvertes
   $$|sessions\,(OE\,(U))| \leq N$$
3. contrainte pour une relation **session - rôle**, par exemple :
   (a) deux rôles en conflit ne peuvent pas être activés dans la même session
   $$|roles\,(OE\,(S)) \cap OE\,(CR)| \leq 1$$
4. contrainte pour une relation d'héritage **rôle - rôle**, par exemple :
   (a) un utilisateur ne peut être attaché à des rôles seniors qui héritent de rôles juniors en conflit
   $$|roles^{\star}\,(OE\,(U)) \cap OE\,(CR)| \leq 1$$
5. contrainte pour une relation **rôle - fonction**, par exemple :

(a) une fonction f_i ne peut pas être affectée à deux rôles en conflit

$$|roles\,(f_i) \cap OE\,(CR)| \leq 1$$

(b) un rôle ne peut pas contenir des fonctions définies comme conflictuelles

$$functions\,(OE\,(R)) \cap OE\,(CF) = \emptyset$$

6. contrainte pour une relation d'héritage **fonction - fonction**, par exemple :

(a) un rôle ne peut être attaché à des fonctions seniors qui héritent de fonctions juniors en conflit

$$|functions^{\star}\,(OE\,(R)) \cap OE\,(CF)| \leq 1$$

7. contrainte pour une relation **fonction - permission**, par exemple :

(a) une permission p_i ne peut pas être affectée à deux fonctions en conflit

$$|functions\,(p_i) \cap OE\,(CF)| \leq 1$$

(b) une fonction ne peut pas avoir des permissions définies comme conflictuelles

$$permissions\,(OE\,(F)) \cap OE\,(CP) = \emptyset$$

On a présenté en utilisant le langage RCL 2000 étendu, des exemples de types de contrainte entre les différents éléments du modèle RBAC étendu. Les contraintes sur les relations "utilisateur-session" et "session-rôle" représentent des contraintes dynamiques. Le contraintes sur les autres types de relations sont des contraintes statiques.

Les contraintes sur les relations "utilisateur-rôle" sont gérées uniquement par l'administrateur de sécurité. Les contraintes sur les autres types de relations peuvent être gérées par le concepteur ou par l'administrateur de sécurité.

4.2.5 Confrontation des points de vues et incohérences

Nous avons précisé, dans le Chapitre 3, deux étapes principales dans le processus de réalisation de sécurité des systèmes d'information : une étape de conception d'une application réalisée par le concepteur et une étape d'exploitation effectuée par l'administrateur de sécurité.

Le concepteur et l'administrateur de sécurité ont des responsabilités différentes. Le concepteur doit répondre aux besoins des futurs utilisateurs du système. L'administrateur de sécurité doit prendre soin de l'affectation des rôles aux utilisateurs en respectant la politique de sécurité globale.

La confrontation de deux points de vues consiste à vérifier que le travail du concepteur ne contient pas d'incohérences par rapport au travail de l'administrateur de sécurité. Cette confrontation va permettre de mettre en lumière les problèmes éventuels qui peuvent apparaître entre les deux niveaux. Ces problèmes peuvent être provoqués pour les raisons suivantes :

- ajout d'une nouvelle fonction d'entreprise dans un système par l'administrateur de sécurité,
- ajout d'une nouvelle application dans un système qui utilise les éléments existants issus d'autres applications du système,
- ajout d'une nouvelle application avec de nouveaux rôles, fonctions et permissions qui seront ensuite introduits dans une hiérarchie de rôles existante,
- suppression d'élément, i.e. fonction d'entreprise, application, rôles d'une application, etc., dans le système par l'administrateur.

Il faudra donc trouver des solutions pour résoudre ces problèmes (section 4.3.2). Nous présentons des situations dans lesquelles ces conflits peuvent apparaître et des solutions possibles pour y remédier.

4.2.5.1 Ajout d'une nouvelle fonction

On veut ajouter une nouvelle fonction d'entreprise dans le système. Par exemple, un administrateur veut ajouter dans le système d'information qui contient l'application "Gestion des Notes" une nouvelle fonction d'entreprise - *Responsable des étudiants*. La personne qui peut être attachée à cette fonction est responsable des étudiants, elle peut lire les informations concernant les étudiants et en particulier les notes de ces étudiants, mais elle ne peut pas changer ces informations. Le *Responsable des étudiants* peut imprimer et visualiser les bulletins de notes, la liste des notes et la liste des notes des étudiants pour une épreuve. Il a les mêmes responsabilités et activités qu'une partie des responsabilités et activités d'un rôle *Secrétariat* qui existe déjà dans le système.

La définition des droits, l'affectation des éléments pour la nouvelle fonction d'entreprise peut être faite au niveau du concepteur ou au niveau de l'administrateur de sécurité.

Les problèmes éventuels qui peuvent apparaître entre le niveau du concepteur et le niveau de l'administrateur de sécurité concernent avant tout les situations dans lesquelles l'administrateur ajoute des nouvelles fonctions pour lesquelles il n'y a pas de rôles précis définis par le concepteur.

Les modifications qui peuvent résoudre cette situation peuvent être divisées en deux approches :

1. L'application est retournée au concepteur pour la réorganiser - l'administrateur change le cahier des charges en ajoutant la nouvelle fonction d'entreprise.
   Le concepteur ajoute les nouveaux éléments et les relations entre ces éléments et les éléments qui existent déjà dans l'application. Le concepteur peut aussi définir de nouvelles contraintes pour les nouveaux éléments et les nouvelles relations. L'application est retournée à l'administrateur de sécurité.
   Dans notre exemple, il ajoute le nouveau rôle *ResponsableDesEtudiants* et attache les fonctions nécessaires à ce rôle, par exemple la fonction *visualisation*.
2. L'administrateur attache cette nouvelle fonction d'entreprise à l'un des rôles existant qui est le plus proche des responsabilités et des droits associés de cette fonction. Il doit définir les contraintes qui limiteront les activités de la nouvelle fonction d'entreprise à celles nécessaires pour elle.
   Dans notre exemple du *Responsable des Étudiants* l'administrateur lui donne seulement les droits de visualiser les informations concernant les étudiants ; il définit les contraintes qui limitent ces droits aux éléments suivants :

$$O_{ResponsableDesEtudiants} = \{Etudiant,\ Matiere,\ Epreuve,\ Controle,\ Note\}$$

$$M_{ResponsableDesEtudiants} = \{visualiser\ des\ notes,\ visualiser\ un\ bulletin,\ ...\}$$

$$P_{ResponsableDesEtudiants} = O_{ResponsableDesEtudiants} \times M_{ResponsableDesEtudiants}$$

$$F_{ResponsableDesEtudiants} = \{visualiser\}$$

$$R_{ResponsableDesEtudiants} = \{Secretariat\}$$

4.2.5.2 Ajout d'une nouvelle application qui utilise les éléments d'autres applications

D'autres problèmes peuvent apparaître quand on ajoute une nouvelle application dans un système d'information et que cette application utilise des éléments, i.e. des données qui existent déjà dans le système d'information.

Par exemple, dans un système d'information il existe entre autres l'application "Gestion des Etudiants" et on ajoute la nouvelle application "Gestion des Notes". L'application "Gestion des Etudiants" contient les informations des étudiants (par exemple : le nom, le prénom, l'adresse, la date de naissance, l'année des études, les cours choisis, etc.). L'application "Gestion des Notes" s'occupe des notes des étudiants des différentes matières. Elle utilise les données de l'application "Gestion des Etudiants" comme le nom ou le prénom d'un étudiant.

Dans ce cas, l'application peut être retournée au niveau du concepteur pour éliminer les incohérences. Cependant, la solution la plus adaptée sera apportée par l'administrateur de sécurité

qui est le mieux placé pour traiter les incohérences éventuelles qui apparaissent entre les deux applications au niveau du système.

L'administrateur de sécurité doit déterminer les données (les objets) de la première application qui peuvent être utilisées par la deuxième application. Il définit que seules les informations comme, par exemple le nom et le prénom de chaque étudiant peuvent être accessibles par les rôles de la deuxième application

$$O_{GestionEtudiants \times GestionNotes} = \{nomEtudiant,\ prenomEtudiant\}$$

L'administrateur de sécurité doit définir les contraintes qui vont déterminer les ensembles des objets de la première application qui peuvent être utilisés par la deuxième application.

4.2.5.3 Ajout de nouveaux rôles dans la hiérarchie des rôles

L'ajout de rôles d'une nouvelle application au sein d'une hiérarchie de rôles peut provoquer aussi des conflits au niveau de l'administrateur. L'introduction de ces nouveaux rôles peut causer des liaisons indirectes entre les rôles conflictuels.

Ces incohérences peuvent être résolues au niveau de l'administrateur de sécurité. Supposons que notre système contienne plusieurs applications. Chaque application possède un ensemble d'éléments en conflit, i.e. un ensemble de rôles en conflit

$$CR = \{CR_{A_1}, CR_{A_2}, ...\} \text{ - } \textit{ensemble de rôles en conflit du système existant}$$
$$CR_{A_i} = \{cr_{A_{i_1}}, cr_{A_{i_2}}, ...\} \text{ - } \textit{ensemble de rôles en conflit de l'application } A_i$$

Supposons que l'on ajoute une nouvelle application avec de nouveaux éléments. On veut introduire les nouveaux rôles dans la hiérarchie de rôles du système

$$R_{A_n} = \{R_a, R_b, R_c, R_d, ...) \text{ - } \textit{ensemble de rôles de l'application } A_n$$

L'introduction de nouveaux rôles dans une hiérarchie de rôles peut se rapporter au deux cas de la Figure 4.5 :

- On ajoute par exemple, le rôle R_b par l'établissement d'une relation entre le rôle R_b et le rôle R_4. Ce rôle R_4 peut avoir des rôles fils qui sont en conflit avec le rôle R_b. Donc il faut vérifier si on peut établir cette relation. Il faut parcourir tous les rôles fils du rôles R_4 et trouver éventuellement les rôles conflictuels avec R_b. Soit $ensembleRolesFils\ (r_i)$ la fonction qui retourne l'ensemble des rôles fils pour le rôle r_i. Alors

$$eR_4 = ensembleRolesFils\ (R_4) - \textit{ensemble de roles fils pour } R_4$$

$$si \quad \forall\, cr_i \in CR\ (R_b \in cr_i \wedge (cr_i \cap eR_4 = \emptyset)) \qquad (1)$$

  alors l'ensemble eR_4 ne contient pas de rôles conflictuels avec le rôle R_b et l'établissement de cette relation est possible.
- On ajoute deux rôles R_a et R_b dans une hiérarchie et on les attache à deux autres rôles qui s'excluent mutuellement R_7, R_9 et qui existent déjà dans cette hiérarchie. Ensuite, on lie les rôles R_a et R_b par la relation d'héritage. Cette relation provoque une relation d'héritage indirecte entre les rôles R_7 et R_9 qui sont définis comme des rôles en conflit. Donc, il existe un conflit dans la hiérarchie de rôles du système.
  Pour vérifier si on peut établir la relation d'héritage entre les rôles R_a et R_b, il faut parcourir la hiérarchie de rôles et trouver les rôles fils du rôle R_a et du rôle R_b :

$$eR_a = ensembleRolesFils\ (R_a) - \textit{un ensemble de roles fils pour } R_a$$

$$eR_b = ensembleRolesFils\ (R_b) - \textit{un ensemble de roles fils pour } R_b$$

$$si \quad \forall\, cr_i \in CR\ ((cr_i \cap eR_a = \emptyset) \wedge (cr_i \cap eR_b = \emptyset)) \qquad (2)$$

  alors les ensembles eR_a et eR_b ne contiennent pas de rôles conflictuels, donc la liaison directe entre les rôles R_a et R_b peut être établie. Dans le cas contraire, la relation entre les rôles R_a et R_b va provoquer une liaison indirecte entre les rôles R_7 et R_9 qui est interdite.

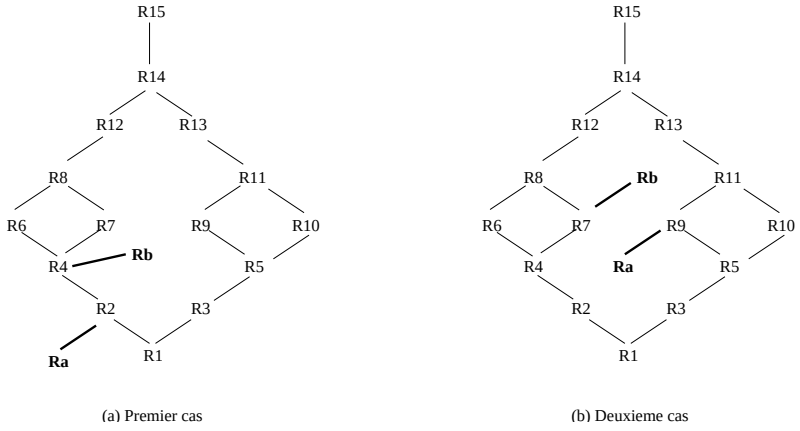


FIG. 4.5 – L'exemple d'une hiérarchie de rôles du système.

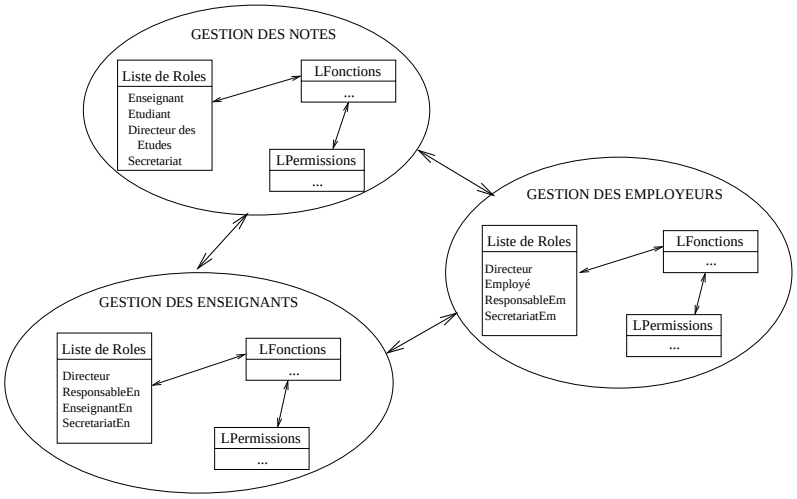


FIG. 4.6 – Les applications du système et la nouvelle application.

La possibilité d'établissement de ces relations entre les rôles peut apparaître si les contraintes définies pour les rôles par le concepteur ou par l'administrateur de sécurité sont changées. Cette situation nécessite de redéfinir les contraintes par le concepteur ou par l'administrateur.

Nous allons illustrer ce cas par un exemple. Soit le système d'information d'un département d'enseignement. Il y a plusieurs applications qui travaillent dans le même système, entre autre l'application "Gestion des Employés", l'application "Gestion des Enseignants". On ajoute une nouvelle application qui s'appelle "Gestion des Notes" (Figure 4.6).

La Figure 4.7(a) présente la hiérarchie de rôles avant l'ajout de la nouvelle application. L'ajout de la nouvelle application nécessite l'ajout de nouveaux rôles dans une hiérarchie de rôles (il faut ajouter dans cet exemple les rôles suivants : *Enseignant*, *Secrétariat*, ...).

Les deux rôles *Enseignant* et *Secrétariat* sont définis dans le système comme des rôles qui s'excluent mutuellement et ils ne peuvent pas être liés par la relation d'héritage. Ces nouveaux rôles peuvent être en relations d'héritage avec les autres rôles qui existent déjà dans le système. Supposant que l'administrateur de sécurité établit les relations entre les nouveaux rôles et les rôles existants comme dans la Figure 4.7(b). Il crée les relations d'héritage entre le rôle *Enseignant* et les rôles *Employé*, *EnseignantEn*, et entre le rôle *Secrétariat* et les rôles *SecrétariatEn*, *Directeur*.

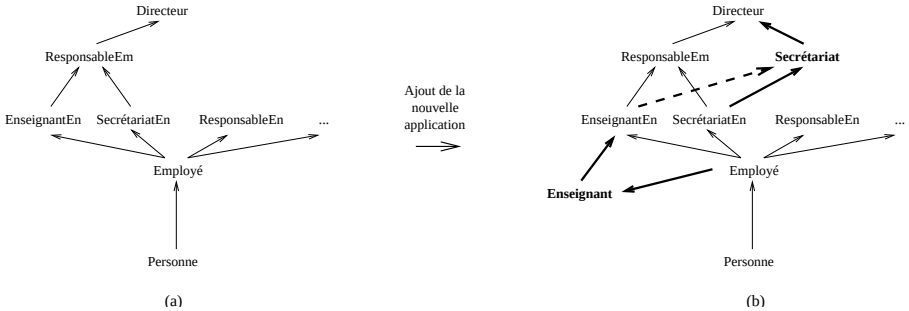


FIG. 4.7 – Les relations entre les rôles.

Ensuite, l'administrateur établit la relation d'héritage entre le rôle *EnseignantEn* (rôle de fils) et rôle *Secrétariat* (rôle de père). Cela peut provoquer des incohérences, parce que les deux rôles *Enseignant* et *Secrétariat* seront maintenant liés indirectement par la relation d'héritage - le rôle *Enseignant* sera hérité par le rôle *Secrétariat*.

Deux solutions sont possibles. On peut ajouter une contrainte qui interdit l'ajout des rôles *Enseignant* et *Secrétariat* comme présenté précédemment. Une autre contrainte peut permettre cet ajout des rôles mais interdire les trois relations présentées.

Cette contrainte est définie par un administrateur de sécurité, elle peut être précisée comme une contrainte au niveau de l'administration.

4.2.5.4 Suppression d'élément

Les modifications du système d'information comme la suppression d'une application ou d'une fonction d'entreprise sont de la compétence de l'administrateur de sécurité et elles sortent du cadre de la confrontation des points de vue.

On a présenté dans cette section la confrontation de deux points de vue différents. Les tâches du concepteur et de l'administrateur de sécurité sont différentes, mais le travail de ce dernier a lieu après le travail du premier. L'administrateur de sécurité travaille sur les résultats du concepteur pour l'intégrer dans le cadre de la politique globale de sécurité du système d'information.

4.3 La chaîne d'intégration basée sur les contraintes

On présente dans cette section l'intégration de nouveaux éléments dans un système en se basant sur le respect des contraintes pour assurer la cohérence globale du système d'information.

Pour permettre une validation plus ou moins automatique, il est nécessaire d'uniformiser les concepts utilisés par le concepteur et l'administrateur de sécurité. Pour cela nous allons mettre en place une étape de traduction. Les étapes de la chaîne d'intégration sont donc les suivantes :

1. traduction du Modèle d'UML réalisé par le concepteur dans un modèle compris par l'administrateur de sécurité,
2. intégration des nouveaux éléments en accord avec les contraintes définies par le concepteur (au niveau d'une application) et par l'administrateur de sécurité (au niveau d'un système d'information qui va contenir cette application).

4.3.1 L'étape de traduction

Le travail du concepteur est utilisé par l'administrateur de sécurité et non l'inverse. Donc il est judicieux de traduire le travail du concepteur dans un langage compris par l'administrateur. Pour cette raison, nous proposons l'approche suivante :

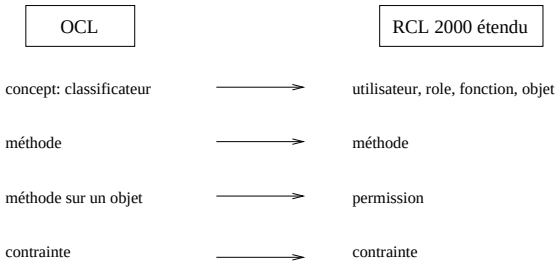


FIG. 4.8 – Le rapprochement des concepts d'OCL et de RCL 2000 étendu.

1. le concepteur crée un Modèle dans le langage UML avec des contraintes en OCL,
2. le parseur traduit le modèle UML (avec les contraintes en OCL) vers un modèle RBAC étendu (avec les contraintes en RCL 2000 étendu),
3. l'administrateur de sécurité intègre les nouveaux éléments dans le modèle du système existant.

La possibilité de traduire le langage OCL en langage RCL 2000 étendu sera donnée pour les trois types de contraintes qui ont été identifiées au niveau de la conception : les contraintes sur une permission, les contraintes de pré-requis et les contraintes de cardinalité. On illustrera cette traduction par des exemples de contraintes issus de l'application "Gestion des Notes".

4.3.1.1 La traduction des concepts communs

Il faut identifier les concepts à traduire et pour chaque concept indiquer la méthode de traduction. La Figure 4.8 propose un rapprochement entre les concepts manipulés en OCL et ceux utilisés en RCL 2000 étendu.

On peut manipuler dans le langage OCL le concept du *classificateur*. Les différents types de classificateurs sont les éléments du langage UML [KMPRS98] : un *acteur (actor)*, un *cas d'utilisation (use case)*, une *classe (class)*, un *objet (object)*, un *attribut (attribute)* d'un objet, une *méthode (method)* d'un objet et une *interface*.

Suivant la nature du classificateur, les concepts correspondants en RCL 2000 étendu sont différents :

- un classificateur de type *Actor* est représenté comme un *Rôle* dans le langage RCL 2000 étendu : $Actor \rightarrow Role$,
- un classificateur de type *Use Case* est représenté comme une *Fonction* dans le langage RCL 2000 étendu : $UseCase \rightarrow Fonction$,
- un classificateur de type *Class* est représenté comme une *Classe* dans le langage RCL 2000 étendu : $Class \rightarrow Classe$,
- un classificateur de type *Object* est représenté comme un *Objet* : $Object \rightarrow Objet$,
- une propriété *Method* d'un classificateur de type *Classe* est représentée comme une *Methode* : $Method \rightarrow Methode$,
- la méthode *getName()* de l'interface *Personne* en OCL est représentée par un *Utilisateur* (section 4.2.2.1) dans le langage RCL 2000 étendu : $Personne.getName() \rightarrow Utilisateur$.

Dans les sections suivantes nous proposons une méthode de traduction en RCL 2000 pour chacune des contraintes exprimées en OCL.

4.3.1.2 La traduction des contraintes sur une permission

En OCL. Une contrainte sur une permission a pour résultat la réduction de l'ensemble d'objets sur lesquels on peut exécuter la méthode. Les contraintes sur une permission peuvent être

exprimées en OCL sous forme d'invariant, sous forme de pré-condition ou sous forme de post-condition. Les pré-conditions donnent la possibilité de représenter des contraintes qui doivent être satisfaites avant d'avoir l'accès aux objets.

Nous avons présenté dans la section 4.1.1.2 la manière de représenter une contrainte de type pré-condition :

context Classe : : Methode
pre : objet | objetCondition.

Une condition *objetCondition* est exprimée par une expression booléenne. La représentation détaillée de cette condition *objetCondition* est donnée dans l'Annexe C.

Par exemple la contrainte : *un Etudiant peut visualiser seulement ses propres notes* sera écrite en OCL comme suit :

$$context\ Note\ ::\ getValeur()\ :\ double$$
$$pre\ :\ n \mid n.oclIsTypeOf\,(Note)\ and\ n.etudiant.getNom() = PEtudiant.getName()$$

où $PEtudiant.getName()$ représente la personne connectée au système.

En RCL 2000 étendu. Le langage RCL 2000 étendu dispose d'ensembles définis (U, R, F, P, M, O, OP, C et S) et de fonctions (section 4.1.2). En utilisant le langage RCL 2000 on peut écrire :

$$p\,(m,\ o) = OE\,(P)$$

Pour cette permission il faut déterminer un ensemble d'objets sur lesquels un utilisateur peut exécuter la méthode. L'ensemble d'objets est décrit en général comme suit :

$$O' = \{o \mid class\,(o) = C\ \wedge\ expressionB\,(o) = vrai\}$$

expressionB - une expression booléenne qui représente le prédicat π (section 4.2.2.1)

L'ensemble O' contient les objets qui sont de la classe C et pour lesquels la condition $expressionB\,(o)$ est vraie. Cette condition peut se composer de plusieurs sous-conditions et cela de manière récursive. L'évolution de cette expression booléenne jusqu'aux éléments de base est présentée dans l'Annexe C.

D'OCL à RCL 2000 étendu

Il faut traduire les expressions d'OCL en RCL 2000 étendu, c'est-à-dire traduire tous les éléments que nous avons énumérés dans la définition de la contrainte en OCL :

- une classe - *Classe*,
- une méthode - *Methode*,
- un objet - *objet*,
- une condition sur l'objet - *objetCondition*.

Les concepts de classe, de méthode et d'objet en OCL sont identiques aux concepts de classe, de méthode et d'objet dans le langage RCL 2000 étendu. On les traduit directement comme suit :

$$traductionClasse\,(Classe\ en\ OCL) \rightarrow c \in C\ en\ RCL2000$$

$$traductionMethode\,(Methode\ en\ OCL) \rightarrow m \in M\ en\ RCL2000$$

$$traductionObjet\,(objet\ en\ OCL) \rightarrow o \in O\ en\ RCL2000$$

où *traductionClasse*, *traductionMethode* et *traductionObjet* représentent les méthodes de traduction d'une classe, d'une méthode et d'un objet de OCL en RCL 2000 étendu.

Une condition sur l'objet *objetCondition* qui détermine une contrainte en OCL peut être traduite en un prédicat exprimé par une expression booléenne en RCL 2000 étendu :

$$traductionCondition\,(objetCondition\ en\ OCL) \rightarrow expressionB()\ en\ RCL2000$$

La traduction est présentée en détails dans l'Annexe C.
La présentation d'une contrainte sur une permission en RCL 2000 étendu est la suivante ;

$$object\,(OE\,(P)) = object\,(p\,(m, o)) = O'$$

Un exemple de contrainte présentée au début de cette section : *un Etudiant peut visualiser seulement ses propres notes* est donné en RCL 2000 étendu :

$$object\,(p\,(getValeur(), note)) = O' = \{o \mid class\,(o) = Note \ \wedge \ o.etudiant.getNom() = user\,(s_i)\}$$

4.3.1.3 La traduction des contraintes de pré-requis

En OCL. Les contraintes de pré-requis sont exprimées en OCL sous forme d'invariant. La définition de telle contrainte nous donne la condition qui doit être satisfaite pour pouvoir exécuter une action. Nous avons présenté dans la section 4.1.1.2 la manière de représenter une contrainte de type invariant :

context Classe inv :
Condition

Cette forme de l'expression *Condition* peut être décomposée en deux parties :

$$Condition = Condition1\ implies\ Condition2$$

La classe *Classe* est un élément qui peut être de type :

$$Classe ::= acteur \mid use\ case \mid methode \mid objet$$

Une description détaillée de ces éléments est présentée dans l'Annexe C.
Par exemple, une contrainte de pré-requis : *la permission de lire un fichier oblige d'avoir la permission de lire le répertoire dans lequel ce fichier est situé* est présentée en OCL comme suit :
context SystemeFichiers inv :

$$self.permission \rightarrow includes\,(p\,('lire',\ 'fichier'))\ \ implies$$

$$self.permission \rightarrow includes\,(p\,('lire',\ 'fichier.repertoire'))$$

En RCL 2000 étendu. Les éléments qui sont importants dans ce type de contrainte sont les rôles, les fonctions et les permissions. Ils peuvent être obtenus par les fonctions du langage RCL 2000 étendu : ***roles*** et ***fonctions***. La *permission* a été définie dans la section 4.3.1.2.

D'OCL à RCL 2000 étendu
Il faut traduire les éléments que nous avons énumérés dans la définition de la contrainte de pré-requis en OCL :
- un acteur - *Acteur*,
- un cas d'utilisation - *UseCase*,
- une méthode - *Methode*,
- un objet - *objet*.

La traduction d'un acteur *Acteur*, d'un cas d'utilisation *UseCase*, d'une méthode *Methode* et d'un objet *objet* du langage OCL en RCL 2000 étendu est la suivante :

$$traductionActeur\,(Acteur\ en\ OCL) \rightarrow r \in R\ en\ RCL2000$$

$$traductionUseCase(UseCase\ en\ OCL) \rightarrow f \in F\ en\ RCL2000$$

$$traductionMethode\,(Methode\ en\ OCL) \rightarrow m \in M\ en\ RCL2000$$

$$traductionObjet\,(objet\ en\ OCL) \rightarrow o \in O\ en\ RCL2000$$

où *traductionActeur*, et *traductionUseCase* représentent les méthodes de traduction d'un acteur et d'un cas d'utilisation d'OCL en RCL 2000 étendu.

Il faut lier les concepts d'OCL avec ceux de RCL 2000 étendu :
- pour les fonctions pré-requis :
  OCL :
  context Acteur inv :
  $self.UseCase \rightarrow includes\,(uc1)\ implies\ self.UseCase \rightarrow includes\,(uc2)$
  RCL 2000 étendu : $f1 \in functions\,(OE\,(R))\ \Rightarrow\ f2 \in functions\,(OE\,(R))$
- pour les permissions pré-requis :
  OCL :
  context UseCase inv :
  $self.permission \rightarrow includes\,(p1)\ implies\ self.permission \rightarrow includes\,(p2)$
  ou
  context UseCase inv :
  $self.permission \rightarrow includes\,(p\,(methode1,\ objet1))\ implies$
  $$self.permission \rightarrow includes\,(p\,(methode2,\ objet2))$$
  RCL 2000 étendu : $p1 \in permissions\,(OE\,(F))\ \Rightarrow\ p2 \in permissions\,(OE\,(F))$ ou
  $p\,(methode1,\ objet1) \in permissions\,(OE\,(F)) \Rightarrow p\,(methode2,\ objet2) \in permissions\,(OE\,(F))$

L'exemple de contrainte de pré-requis présenté au début de cette section : *la permission de lire un fichier oblige d'avoir la permission de lire le répertoire dans lequel ce fichier est situé* peut être traduite en RCL 2000 étendu comme suit :

$$p\,('lire',' fichier') \in permissions\,(OE\,(F)) \Rightarrow p\,('lire',' fichier.repertoire') \in permissions\,(OE\,(F))$$

4.3.1.4 La traduction des contraintes de cardinalité

En OCL. Les contraintes de cardinalité sont également exprimées en OCL sous forme d'invariants. Nous avons présenté dans la section 4.1.1.2 la représentation d'une contrainte de type invariant :

context Classe inv :
Condition

Cette forme de l'expression *Condition* peut être présentée comme suit :

$$Condition ::= elementCondition\ implies\ numCondition$$

Les conditions *elementCondition* et *numCondition* représentent des expressions booléennes. La représentation de ces expressions en OCL est la même que la condition *objetCondition* déterminée dans la section 4.3.1.2. Une description détaillée de ces conditions est donnée dans l'Annexe C.

Par exemple, la contrainte de cardinalité : *la permission de lire le fichier des mots de passe du système doit être associée seulement à un rôle*, est décrite en OCL comme suit :
context Permission inv :
$(self.methode \rightarrow includes\,('lire')\ and\ self.objet \rightarrow includes\,('MotsDePasse'))$
$$implies\ self.cas\ d'utilisation.acteur = 1$$

En RCL 2000 étendu. Les éléments qui sont importants dans ce type de contrainte sont les rôles, les fonctions, les permissions, les méthodes et les objets. Ils peuvent être obtenus par les fonctions de RCL 2000 étendu correspondantes.

D'OCL à RCL 2000 étendu

Il faut traduire tous les éléments que nous avons énumérés dans la définition de la contrainte en OCL :

- une classe - *Classe*,
- une condition sur les objets de la classe - *elementCondition*,
- une condition numérique - *numCondition*.

La traduction d'une classe *Classe* d'OCL en RCL 2000 étendu est obtenue par :

$$traductionClasse\,(Classe\ en\ OCL) \rightarrow c \in C\ en\ RCL2000$$

La traduction des conditions *elementCondition* et *numCondition* peut être réalisée de la même manière que la traduction d'une condition sur l'objet *objetCondition* présentée dans la section 4.3.1.2. Donc, on traduit ces conditions vers un prédicat exprimé par une expression booléenne en RCL 2000 étendu :

$$traductionCondition\,(elementCondition\ en\ OCL) \rightarrow expressionB()\ en\ RCL2000$$

$$traductionCondition\,(numCondition\ en\ OCL) \rightarrow expressionB()\ en\ RCL2000$$

La traduction détaillée de ces conditions *elementCondition* et *numCondition* est présentée dans l'Annexe C.

L'exemple de contrainte de cardinalité présentée au début de cette section : *la permission de lire le fichier des mots de passe du système doit être associée seulement à un rôle* peut être décrite en RCL 2000 étendu comme suit :

$$|roles\,(OE\,(functions\,(p\,('lire',\ 'MotsDePasse'))))| = 1$$

4.3.1.5 L'illustration des méthodes de traduction sur notre exemple

On présente des exemples de contraintes du point de vue du concepteur d'une application. La Figure 4.9 présente un extrait du diagramme de classes d'une application "Gestion des Notes" dans laquelle nous allons ajouter des contraintes. Ce diagramme contient les classes, l'interface et la classe association (Note).

On commence par la contrainte sur une permission :

un Etudiant peut visualiser uniquement ses propres notes

Cet exemple de contrainte se rapporte à un diagramme de séquence "visualiser des notes" dans lequel un étudiant demande un accès pour visualiser (lire) ses notes (Figure 4.10).

Un utilisateur qui s'appelle Robert et qui joue le rôle *Etudiant* (un acteur *Etudiant*) veut visualiser ses notes. Il exécute le cas d'utilisation *visualisation des notes* qui est décrit par le diagramme de séquence *"visualisation des notes"*. Il doit regarder la liste de toutes ses matières - il exécute la méthode *getMatière(Etudiant)* qui retourne la liste des matières auxquelles il a participé (un accès à un objet de la classe *listeMatière*). Il choisit la matière qu'il veut regarder et exécute la méthode *getContrôle(Etudiant, Matière)*. Il obtient la liste des contrôles faits pour la matière choisie (un accès à un objet de la classe *ListeContrôle*). Pour le *contrôle* choisi (un objet de la classe *Contrôle*) il exécute la méthode *getNote(Etudiant, Matière, Contrôle)*. Le résultat de cette exécution est un ensemble de valeurs de notes qui dépend de l'étudiant qui a lancé l'exécution.

Cette exécution doit respecter la contrainte qui dit que l'étudiant ne peut visualiser que ses propres notes. Elle peut être représentée de la manière suivante :

$$context\ Note\ ::\ getValeur()\ :\ double$$
$$pre\ :\ n \mid n.oclIsTypeOf\,(Note)\ and\ n.etudiant.getNom() = PEtudiant.getName()$$

ou encore :

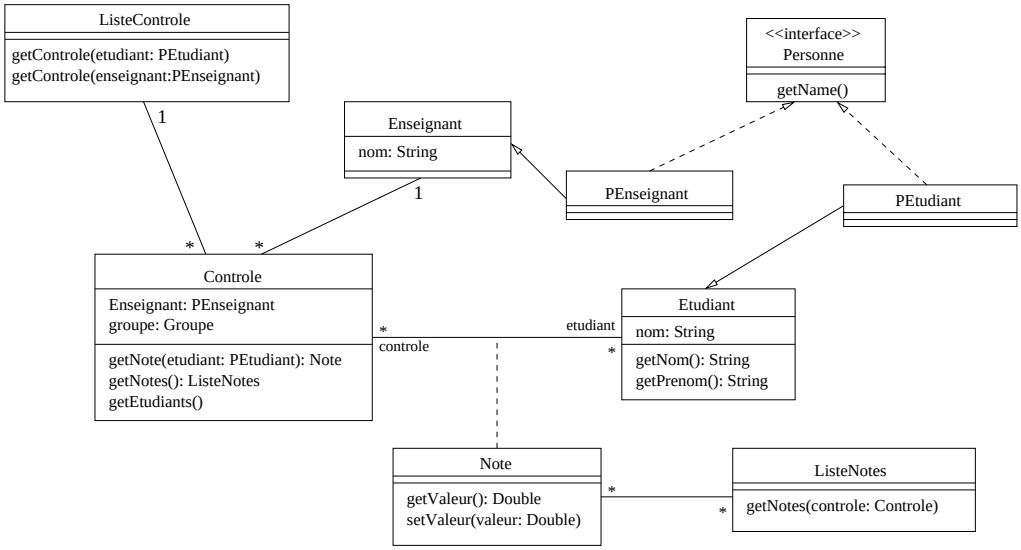


FIG. 4.9 – L'extrait du diagramme de classes de l'exemple d'application "Gestion des Notes".

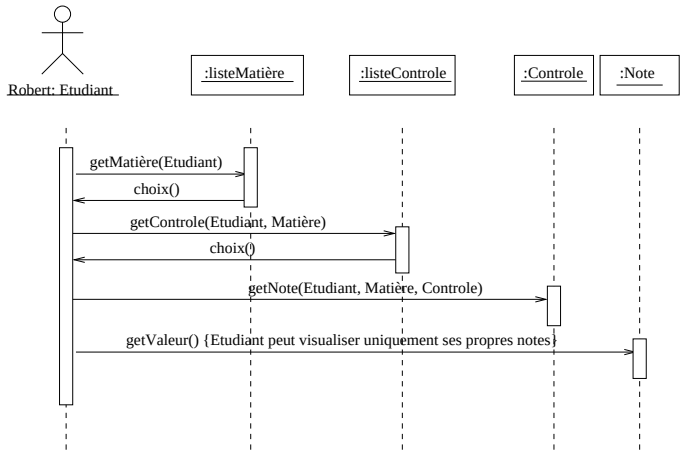


FIG. 4.10 – L'extrait du diagramme de séquence "visualiser des notes".

$context\ Note\ ::\ getValeur()\ :\ double$
$pre\ :\ note \mid note.getNom() = PEtudiant.getName()$

Nous allons montrer la traduction de cette contrainte qui est une contrainte sur une permission. On définit dans cet exemple une contrainte dans la classe *Note* pour la méthode *getValeur()*. Cette contrainte est définie comme une pré-condition. Parmi l'ensemble de tous les instances de la classe *Note* (*note*) on choisit uniquement celles qui vérifient la condition donnée.

La condition dit que l'on choisit les objets de la classe *Note* pour lesquels le propriétaire (défini par son nom - *etudiant.getNom()*, où *etudiant* est le *roleName* de l'association entre la classe *Contrôle* et la classe *Etudiant* et la méthode *getNom()* de la classe *Etudiant* retourne le nom d'un étudiant) est celui qui demande l'accès aux notes. L'identité de cet utilisateur est vérifié par la fonction *getName()* qui vient d'une interface *Personne* qui a été définie pour l'identification d'un utilisateur dans une application.

On peut indiquer la nature des éléments de cette contrainte en utilisant les notations suivantes :

Classe := Note
Methode := getValeur()
objet := note
objetCondition := note.etudiant.getNom() = PEtudiant.getName().

D'OCL à RCL 2000 étendu

Il faut traduire tous les éléments que nous avons énumérés dans la contrainte OCL :

- *note* est un objet - $o = note$,
- *Note* est la classe - $Classe = Note$,
- *etudiant* est une instance de la classe *Etudiant* - $roleName = etudiant$,
- *getNom()* est la méthode de la classe *Etudiant* qui retourne le nom d'un étudiant - $methode = getNom()$,
- *PEtudiant.getName()* retourne le nom de l'utilisateur qui est connecté avec le système; pour exprimer cet utilisateur on peut utiliser la fonction de RCL 2000 - *user(s)* qui donne l'utilisateur connecté avec le système dans la session s_i :

$$traduction\,(PEtudiant.getName()) \rightarrow user\,(s_i)$$

Ayant ces éléments définis, on peut compléter la définition par rapport à la contrainte en OCL :

$object\,(p\,(getValeur(), note)) = O' = \{o \mid class\,(o) = Note\ \wedge\ o.etudiant.getNom() = user\,(s_i)\}$

Cet ensemble O' est un ensemble d'objets *note* de la classe *Note* pour lesquels le propriétaire de ces notes est l'utilisateur de la session actuelle.

La présentation des autres types de contraintes

1. ***contrainte de pré-requis***
   un Etudiant peut visualiser ses notes (exécuter la méthode *getNote(Etudiant)*) *si il a la permission de visualiser la liste de contrôles* (la méthode *getContrôle(Etudiant)*)
   en OCL :
   context visualiser inv :
   $self.permission \rightarrow includes\,('getNote(Etudiant)',\ 'note')\ implies$

   $$self.permission \rightarrow includes\,('getControle(Etudiant)',\ 'controle')$$

   et en RCL 2000 étendu :

   $$p\,('getNote(Etudiant)',\ 'note') \in permissions\,('visualiser')\ \Rightarrow$$

   $$p\,('getControle(Etudiant)',\ 'controle') \in permissions\,('visualiser')$$

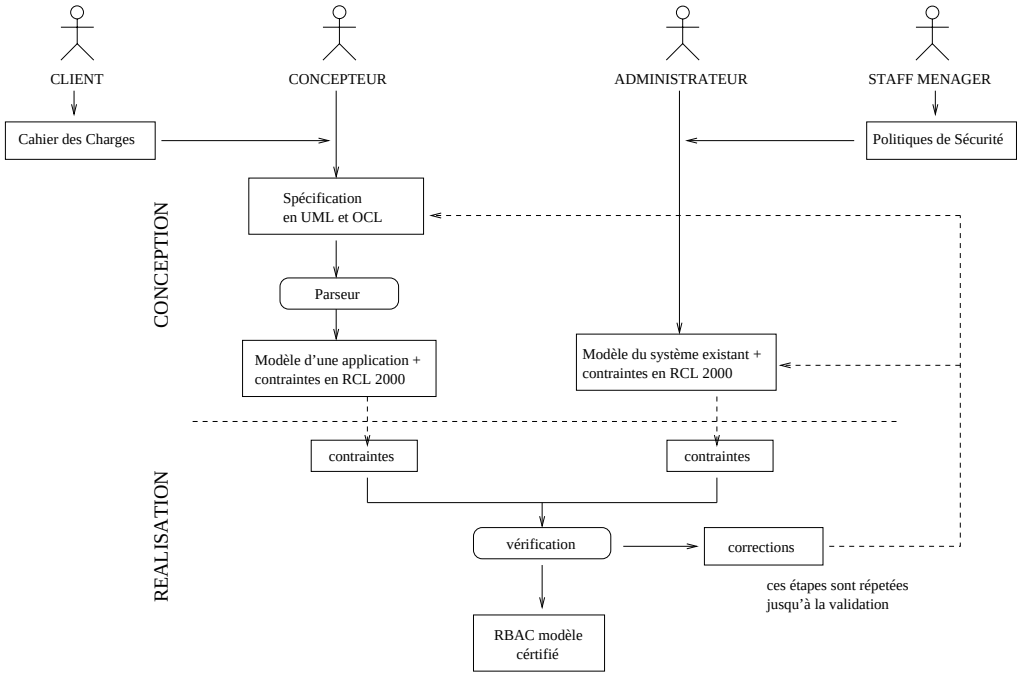


FIG. 4.11 – La validation du concepteur et de l'administrateur.

2. ***contrainte de cardinalité***
   la permission d'imprimer un bulletin de notes doit être associée uniquement à un rôle, par exemple au rôle Directeur des Etudes
   en OCL :
   $context\ Permission\ inv$:
   $(self.methode \rightarrow includes\ ('imprimer')\ and\ self.objet \rightarrow includes\ ('BulletinNotes'))$

   $$implies\ self.cas\ d'utilisation.acteur = 1$$

   en RCL 2000 étendu :

   $$|roles\ (OE\ (functions\ (p\ ('imprimer',\ 'BulletinNotes'))))| = 1$$

Ces exemples illustrent la possibilité de traduire les contraintes exprimées en OCL au niveau de la conception dans le langage RCL 2000 étendu.

4.3.2 La vérification de la cohérence

La vérification a pour but de garantir que l'ajout d'une nouvelle application dans le système d'information ne va pas créer d'incohérences et ceci avant son intégration.

La Figure 4.11 présente le diagramme de flux de la vérification des deux points de vues. La première partie permet de générer la spécification du modèle. Du coté de la conception, le parseur analyse le modèle UML de l'application et conserve les concepts qui seront utilisés pour faire les vérifications. La représentation du modèle est donnée en accord avec le méta-modèle d'UML. Du côté administrateur de sécurité, celui-ci dispose du Modèle RBAC étendu avec les contraintes exprimées en RCL 2000 étendu.

La deuxième partie est une étape de vérification et de correction des éléments et des contraintes du concepteur et de l'administrateur. Il faut comparer ces deux groupes de contraintes et éventuellement accomplir des changements si elles ne sont pas cohérentes.

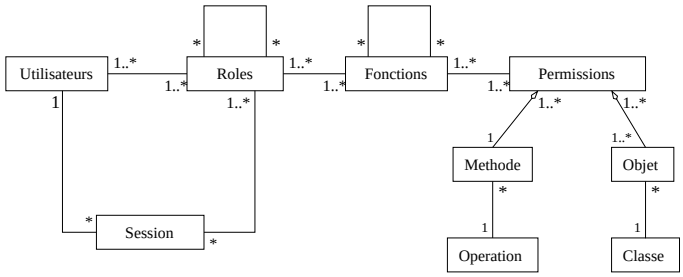


FIG. 4.12 – La cohérence du modèle RBAC étendu.

On réitère les étapes précédentes jusqu'à la validation finale. C'est seulement à ce moment que l'on pourra intégrer la nouvelle application dans le système global.

4.3.2.1 La définition de la cohérence

Le système d'information que l'on utilise doit être cohérent, c'est-à-dire que tous les éléments définis dans le système, leurs relations et leurs contraintes doivent être cohérents. Après avoir donné une définition de la cohérence du système, nous pourrons valider ou non l'intégration d'une nouvelle application dans le système de sécurité existant.

Soit :

E - ensemble des éléments du système qui contient les sous-ensembles : utilisateurs, sessions, rôles, fonctions, permissions, méthodes et objets

$$E = \{U,\ S,\ R,\ F,\ P,\ M,\ O\},\ E_i \in E \text{ et } e_{ij} \in E_i \text{ - un élément d'un ensemble } E_i$$

Les définitions (entre autres les contraintes) déterminées dans le système d'information pour un élément $e_{ij} \in E_i$ limitent un ensemble d'éléments accessibles au niveau du système.

Soit :

$E_S\,(e_{ij})$ - *un ensemble d'éléments en relation avec un élément* e_{ij} *par les associations contraintes dans le système*

$$E_{Sk}\,(e_{ij}) \in E_S\,(e_{ij}),\ E_{Sk}\,(e_{ij}) ::= U_S\,(e_{ij}) \mid S_S\,(e_{ij}) \mid R_S\,(e_{ij}) \mid F_S\,(e_{ij}) \mid P_S\,(e_{ij}) \mid M_S\,(e_{ij}) \mid O_S\,(e_{ij})$$

Nous proposons la définition suivante de la cohérence :

Définition

Le modèle RBAC étendu est ***cohérent*** si et seulement si tout élément $e_{ij} \in E_i$ (utilisateur, rôle, fonction, permission, méthode ou objet) satisfait la condition suivante : *pour chaque élément* $e_{ij} \in E_i$ *l'ensemble des éléments en relation avec* e_{ij} *doit être différent de l'ensemble vide*, i.e.

$$\forall\ e_{ij} \in E_i \wedge \forall k,\ (E_{Sk}\,(e_{ij}) \in E_S\,(e_{ij}) \wedge E_{Sk}\,(e_{ij}) \neq \emptyset)$$

en particulier :

$$\forall\ u_j \in U,\ (E_{Sk}\,(u_j) ::= R_S\,(u_j) \wedge \forall k\ E_{Sk}\,(u_j) \neq \emptyset)$$

$$\forall\ r_j \in R,\ (E_{Sk}\,(r_j) ::= U_S\,(r_j) \mid F_S\,(r_j) \wedge \forall k\ E_{Sk}\,(r_j) \neq \emptyset)$$

$$\forall\ f_j \in F,\ (E_{Sk}\,(f_j) ::= R_S\,(f_j) \mid P_S\,(f_j) \wedge \forall k\ E_{Sk}\,(f_j) \neq \emptyset)$$

$$\forall\ p_j \in P,\ (E_{Sk}\,(p_j) ::= F_S\,(p_j) \mid M_S\,(p_j) \mid O_S\,(p_j) \wedge \forall k\ E_{Sk}\,(p_j) \neq \emptyset)$$

$$\forall\ m_j \in M,\ (E_{Sk}\,(m_j) ::= P_S\,(m_j) \wedge E_{Sk}\,(m_j) \neq \emptyset)$$

$$\forall\ o_j \in O,\ (E_{Sk}\,(o_j) ::= P_S\,(o_j) \wedge E_{Sk}\,(o_j) \neq \emptyset)$$

La Figure 4.12 présente le modèle RBAC étendu cohérent. Pour être en accord avec la définition de la cohérence, nous avons modifié le modèle RBAC étendu présenté dans le Chapitre 4 par rapport à la cardinalité des relations d'association entre les éléments du modèle, i.e. la cardinalité $1..*$ pour les relations : U-R, R-F, F-P.

Justification

L'ensemble des rôles $R_S(u_j)$ accessibles par un utilisateur u_j ne peut pas être vide. En effet un utilisateur qui n'est affecté à aucun rôle ne pourra pas travailler sur le système.

L'ensemble des utilisateurs $U_S(r_j)$ et l'ensemble des fonctions $F_S(r_j)$ accessibles par un rôle r_j dans le système ne peuvent pas être vides. Un rôle doit avoir au moins une fonction attachée pour pouvoir interagir avec le système. Il doit également être attaché au moins à un utilisateur pour pouvoir être activable.

L'ensemble des rôles $R_S(f_j)$ et l'ensemble des permissions $P_S(f_j)$ accessibles par une fonction f_j dans le système ne peuvent pas être vides. Une fonction doit être attachée au moins à un rôle pour être utilisée dans le système. Elle doit avoir au moins une permission attachée pour pouvoir réaliser ses actions.

L'ensemble des fonction $F_S(p_j)$, l'ensemble des méthodes $M_S(p_j)$ et l'ensemble des objets $O_S(p_j)$ accessibles par une permission p_j dans le système ne peuvent pas être vides. Une permission doit être attachée au moins à une fonction pour être utile dans le système. Elle doit au moins posséder une méthode et au moins un objet attaché à cette dernière.

L'ensemble des permissions $P_S(m_j)$ accessibles par une méthode m_j ne peut pas être vide parce que chaque méthode doit être attachée au moins à une permission pour être utilisée dans le système.

L'ensemble des permissions $P_S(o_j)$ accessibles pour un objet o_j dans le système ne peut pas être vide parce que chaque objet doit être attaché au moins à une permission pour accéder à ses informations.

Nous n'avons pas pris en considération le concept d'héritage de rôles et de fonctions (associations réflexives). Dans un ensemble de rôles il existe des rôles qui n'héritent d'aucun autre, donc l'ensemble des rôles $R_S(e_{ij})$ pour $e_{ij} \in R$ peut être vide. De la même façon, il existe des fonctions dans un ensemble des fonctions qui n'héritent d'aucune autre, donc l'ensemble des fonctions $F_S(e_{ij})$ pour $e_{ij} \in F$ peut être vide. En conséquence, la vérification d'un héritage de rôles et d'un héritage de fonctions n'est pas nécessaire pendant la vérification de la cohérence du modèle.

Nous n'avons pas pris en considération le concept de session parce qu'il représente un aspect dynamique du modèle. Une session est ouverte dans le système par un utilisateur qui se connecte avec celui-ci. Une analyse a priori ne peut donc être faire sur les relations U-S et S-R.

La cohérence permet donc de garantir que les éléments issus d'une nouvelle application seront significatifs, c'est-à-dire avec des liaisons vers d'autres éléments du modèle RBAC étendu.

4.3.2.2 Vérification de la cohérence du système

Des incohérences éventuelles peuvent apparaître pendant l'ajout de nouveaux éléments dans un système existant :

1. l'ajout d'un ou plusieurs éléments à l'ensemble des utilisateurs, des rôles, des fonctions, des permissions, des méthodes ou/et des objets,
2. l'ajout d'une ou plusieurs relations entre les éléments du système,
3. l'ajout d'une ou plusieurs contraintes sur les éléments existants dans le système,
4. l'ajout d'une nouvelle application dans le système (i.e. le plus complexe).

Pour prouver que le système après l'ajout des nouveaux éléments est encore cohérent il faut vérifier que la définition de la cohérence (donnée dans la section précédente) reste satisfaite. Dans

la suite, nous allons prendre en considération le cas de l'ajout d'une nouvelle application car il implique l'ajout de nouveaux éléments, de nouvelles relations et de nouvelles contraintes.

Pour trouver des incohérences éventuelles il faut vérifier si l'ensemble des éléments (rôles, fonctions, etc.) de la nouvelle application possède des éléments communs avec l'ensemble des éléments existants. Ces éléments communs, leurs relations avec les autres éléments et leurs contraintes peuvent provoquer des incohérences. Donc, nous proposons de vérifier les conditions de la cohérence pour chaque élément commun.

Soit :

E_A - ensemble des éléments de la nouvelle application qui contient les sous-ensembles : utilisateurs (U_A), sessions (S_A), rôles (R_A), fonctions (F_A), permissions (P_A), méthodes (M_A) et objets (O_A) de la nouvelle application

$$E_A = \{U_A,\ S_A,\ R_A,\ F_A,\ P_A,\ M_A,\ O_A\} \text{ et } E_{Ai} \in E_A$$

Il faut vérifier si l'ensemble

$$E' = E \cap E_A$$

contient des éléments ou non, c'est-à-dire vérifier si la nouvelle application possède des éléments communs avec les éléments déjà présents dans le système. La vérification de la cohérence doit être réalisée pour chaque sous-ensemble de la manière suivante :

$$U' = U \cap U_A,\ R' = R \cap R_A,\ F' = F \cap F_A,\ P' = P \cap P_A,\ M' = M \cap M_A \text{ et } O' = O \cap O_A$$

Soit : $E'_i \in E'$ *et* $E'_i ::= U' \mid S' \mid R' \mid F' \mid P' \mid M' \mid O'$, $ec_{ij} \in E'_i$ - *un élément d'un ensemble* E'_i. Donc, le calcul de l'intersection de l'ensemble E' peut retourner deux résultats :

1. $E' = \emptyset$, c'est-à-dire $E \cap E_A = \emptyset$ - il n'y a pas d'élément commun entre l'ensemble des éléments du système et l'ensemble des éléments de la nouvelle application. Dans ce cas, il n'y a pas d'incohérence entre la nouvelle application et le système existant.
2. $E' \neq \emptyset$, c'est-à-dire $E \cap E_A \neq \emptyset$ - des éléments communs existent et dans ce cas il faut les étudier pour identifier et éliminer les incohérences possibles.

Les contraintes définies pour un élément commun $ec_{ij} \in E'_i$ déterminent un ensemble d'éléments accessibles par lui au niveau de l'application et aussi au niveau du système. Soit :

$E_S\,(ec_{ij})$ - un ensemble d'éléments en relation avec un élément commun ec_{ij} par les associations contraintes dans le système existant

$E_A\,(ec_{ij})$ - un ensemble d'éléments en relation avec un élément commun ec_{ij} par les associations contraintes dans la nouvelle application

$$E_{Sk}\,(ec_{ij}) \in E_S\,(ec_{ij}) \text{ et}$$
$$E_{Sk}\,(ec_{ij}) ::= U_S\,(ec_{ij}) \mid R_S\,(ec_{ij}) \mid F_S\,(ec_{ij}) \mid P_S\,(ec_{ij}) \mid M_S\,(ec_{ij}) \mid O_S\,(ec_{ij})$$

$$E_{Ak}\,(ec_{ij}) \in E_A\,(ec_{ij}) \text{ et}$$
$$E_{Ak}\,(ec_{ij}) ::= U_A\,(ec_{ij}) \mid R_A\,(ec_{ij}) \mid F_A\,(ec_{ij}) \mid P_A\,(ec_{ij}) \mid M_A\,(ec_{ij}) \mid O_A\,(ec_{ij})$$

Exemples

Nous allons montrer deux exemples de situations issus de l'ajout d'une nouvelle application.

Le premier exemple ne provoque pas d'incohérence. Soit un rôle commun $roleA \in R' \subset E'$ et C_1 est la contrainte dans le système telle que :

$$C_1 \ :\ fonctions\,(roleA) \subset \{f_1, f_2, f_3\} \text{ pour } f_1, f_2, f_3 \in F_S$$

La contrainte C_1 définit les fonctions du système qui sont accessibles pour le rôle $roleA$:

$$F_S\,(roleA) \in \{\{f_1, f_2, f_3\}, \{f_1, f_2\}, \{f_1, f_3\}, \{f_2, f_3\}, \{f_1\}, \{f_2\}, \{f_3\}\}$$

Soit C_2 est la contrainte dans la nouvelle application telle que :

$$C_2 \ : \ fonctions\,(roleA) \subset \{f_1, f_2, f_4, f_5\} \ pour \ f_1, f_2, f_4, f_5 \in F_S \cup F_A$$

La contrainte C_2 définit les fonctions du système et/ou de la nouvelle application qui sont accessibles pour le rôle $roleA$:

$$F_A\,(roleA) \in$$

$$\{\{f_1, f_2, f_4, f_5\}, \{f_1, f_2, f_4\}, \{f_1, f_2, f_5\}, \ \ldots \{f_1, f_2\}, \{f_1, f_4\}, \{f_2, f_4\}, \{f_1, f_5\}, \ \ldots \{f_1\}, \{f_2\}, \{f_4\}, \{f_5\}\}$$

Ces deux contraintes C_1 et C_2 limitent l'ensemble des fonctions accessibles pour le rôle $roleA$ au niveau du système et au niveau de la nouvelle application. Elles sont cohérentes, parce que la définition de la cohérence est encore satisfaite après l'ajout de la nouvelle contrainte C_2 :

$$F_S\,(roleA) \cap F_A\,(roleA) = \{\{f_1, f_2\}, \{f_1\}, \{f_2\}\} \neq \emptyset$$

Le deuxième exemple provoque des incohérences. Soit un rôle commun $roleB \in R' \subset E'$ et C_3 la contrainte dans le système :

$$C_3 \ : \ fonctions\,(roleB) \subset \{f_1, f_5, f_6\} \ pour \ f_1, f_5, f_6 \in F_S \wedge |fonctions\,(roleB)| \leq 2$$

La contrainte C_3 définit les fonctions du système qui sont accessibles pour le rôle $roleB$ et limite le nombre de ces fonctions à deux :

$$F_S\,(roleB) \in \{\{f_1, f_5\}, \{f_1, f_6\}, \{f_5, f_6\}, \{f_1\}, \{f_5\}, \{f_6\}\}$$

Soit C_4 la contrainte dans la nouvelle application :

$$C_4 \ : \ fonctions\,(roleB) \supset \{f_1, f_2, f_4\} \ pour \ f_1, f_2, f_4 \in F_S \cup F_A$$

La contrainte C_4 définit les fonctions du système et/ou de la nouvelle application qui sont accessibles pour le rôle $roleB$:

$$F_A\,(roleB) \in \{\{f_1, f_2, f_4, ...\}, \{f_1, f_2, f_4, ...\}, ..., \{f_1, f_2, f_4\}\}$$

Ces deux contraintes, C_3 et C_4, ne sont pas cohérentes, en effet :

$$F_S\,(roleB) \cap F_A\,(roleB) = \emptyset$$

Les sections suivantes présentent comment trouver et éliminer ces incohérences.

4.3.2.3 L'algorithme de vérification de la cohérence du système

Nous avons montré dans la section précédente que l'ajout d'une application dans le système peut provoquer des incohérences. Pour éliminer ces incohérences, nous devons :

1. trouver les éléments communs de chaque type,
2. identifier les contraintes pour chacun des éléments communs,
3. vérifier la cohérence globale des éléments communs,
4. identifier les contraintes qui provoquent des incohérences ,
5. proposer et réaliser des modifications nécessaires pour éliminer ces incohérences.

Trouver les éléments communs de chaque type, c'est-à-dire trouver les utilisateurs communs (U'), les rôles communs (R'), les fonctions communes (F'), les permissions communes (P'), les méthodes communes (M') ou/et les objets communs (O') entre l'application et le système actuel.

Identifier les contraintes pour chacun des éléments communs :

pour un élément $ec_{ij} \in E'_i$: l'ensemble des contraintes contient l'ensemble des contraintes déjà définies pour cet élément dans le système et un ensemble de contraintes définies dans la nouvelle application :

$$\forall\, ec_{ij} \in E'_i,\ C\left(ec_{ij}\right) = C_S\left(ec_{ij}\right) \cup C_A\left(ec_{ij}\right)$$

$$C\left(ec_{ij}\right) - \textit{ensemble des contraintes d'un élément } ec_{ij}$$

$$C_S\left(ec_{ij}\right) - \textit{ensemble des contraintes d'un élément } ec_{ij} \textit{ qui existent dans le système}$$

$$C_A\left(ec_{ij}\right) - \textit{ensemble des contraintes d'un élément } ec_{ij} \textit{ de l'application}$$

En particulier pour les rôles : l'ensemble des contraintes pour un rôle commun $r_j \in R'$ est défini par :

$$\forall\, r_j \in R',\ C\left(r_j\right) = C_S\left(r_j\right) \cup C_A\left(r_j\right)$$

et de la même façon pour les éléments des autres types :

$$\forall\, f_j \in F',\ C\left(f_j\right) = C_S\left(f_j\right) \cup C_A\left(f_j\right)$$

$$\forall\, p_j \in P',\ C\left(p_j\right) = C_S\left(p_j\right) \cup C_A\left(p_j\right)$$

$$\forall\, m_j \in M',\ C\left(m_j\right) = C_S\left(m_j\right) \cup C_A\left(m_j\right)$$

$$\forall\, o_j \in O',\ C\left(o_j\right) = C_S\left(o_j\right) \cup C_A\left(o_j\right)$$

$$\forall\, u_j \in U',\ C\left(u_j\right) = C_S\left(u_j\right) \cup C_A\left(u_j\right)$$

Vérifier la cohérence globale des éléments communs - vérifier que les contraintes ne provoquent pas d'incohérence, c'est-à-dire vérifier que les éléments communs avec leurs contraintes satisfassent la définition de la cohérence :

$$\forall\, ec_{ij} \in E'_i \wedge \forall k,\ \left(E_{Sk}\left(ec_{ij}\right) \in E_S\left(ec_{ij}\right) \wedge E_{Sk}\left(ec_{ij}\right) \neq \emptyset\right)$$

Si les éléments communs sont cohérents, il n'est pas nécessaire de mettre en place des changements dans la nouvelle application. Dans le cas contraire, il faut identifier les contraintes pour chaque élément commun qui provoquent des incohérences.

Identifier les contraintes qui provoquent des incohérences - il faut trouver les contraintes d'un élément commun qui sont responsables d'incohérences sur cet élément. Il faut déterminer quels ensembles de contraintes $C_S\left(ec_{ij}\right)$ et $C_A\left(ec_{ij}\right)$ ne satisfont pas la définition de la cohérence :

$\forall\, ec_{ij} \in E'_i$ *pour lequel des incohérences sont identifiées, trouver les ensembles des contraintes qui sont responsables de ces incohérences, i.e. les ensembles* $C_S\left(ec_{ij}\right)$ *et* $C_A\left(ec_{ij}\right)$.

Proposer et réaliser les modifications nécessaires dans les définitions de contrainte qui provoquent des incohérences. Ces modifications peuvent être effectuées au niveau de l'administrateur de sécurité en accord avec les contraintes définies dans le système et dans la nouvelle application. Si, c'est impossible, il faut les faire au niveau du concepteur de la nouvelle application.

L'algorithme 4.1 montre les étapes de la vérification de cohérence du système après l'ajout d'une nouvelle application.

La première étape, i.e. l'étape d'identification des éléments communs entre le système et la nouvelle application consiste à parcourir les ensembles des éléments du système et de l'application de même type, i.e. U avec U_A, R avec R_A,etc. et à déterminer leurs éléments communs :

$$U' = U \cap U_A,\ R' = R \cap R_A,\ F' = F \cap F_A,\ P' = P \cap P_A,\ M' = M \cap M_A \textit{ et } O' = O \cap O_A$$

L'ensemble des utilisateurs U_A de la nouvelle application sera vide, car le concepteur de l'application ne définit pas les utilisateurs qui utiliseront cette application. Cette tâche est réalisée par l'administrateur de sécurité. C'est lui qui détermine les futurs utilisateurs de l'application dans le système. Donc, dans les sections suivantes nous allons prendre en considération les éléments communs des ensembles :

$$R' = R \cap R_A,\ F' = F \cap F_A,\ P' = P \cap P_A,\ M' = M \cap M_A \textit{ et } O' = O \cap O_A$$

Algorithme 4.1 La vérification de la cohérence du système après l'ajout d'une nouvelle application

$VerificationCoherence\,(E,\ E_A)$

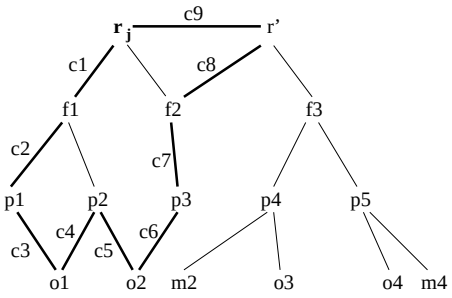
Début
  pour *chaque* $(E_i \in E$ *et* $E_{Ai} \in E_A)$ **faire**
    $E'_i = E_i \cap E_{Ai}$
    si $E'_i \neq \emptyset$ **alors**
      pour *chaque* $ec_{ij} \in E'_i$ **faire**
        $C_S\,(ec_{ij}) = identifierContraintes\,(ec_{ij})$
        $C_A\,(ec_{ij}) = identifierContraintes\,(ec_{ij})$
        $coherence = verifierCoherence\,(ec_{ij},\ C_S\,(ec_{ij})\ ,\ C_A\,(ec_{ij}))$
        si (! cohérence) **alors**
          $CI\,(ec_{ij}) = trouverContraintesIncoherentes\,(ec_{ij},\ C_S\,(ec_{ij})\ ,\ C_A\,(ec_{ij}))$
          $afficher\,(CI\,(ec_{ij}))$
          retourner cohérence
        finSi
      fait
    finSi
  fait
  retourner vrai
Fin

4.3.2.4 L'étape d'identification des contraintes des éléments communs

On veut trouver l'ensemble des contraintes définies pour un élément commun $ec_{ij} \in E'_i$, i.e. un ensemble $C\,(ec_{ij}) = C_S\,(ec_{ij}) \cup C_A\,(ec_{ij})$. On peut identifier pour chaque élément $ec_{ij} \in E'_i$ deux groupes de contraintes au niveau du système et au niveau de la nouvelle application :

- les contraintes qui sont attachées "directement" à l'élément ec_{ij}, i.e. les contraintes se rapportent exactement à cet élément,
- les contraintes qui sont attachées "indirectement" à l'élément ec_{ij}, i.e. les contraintes définies sur les éléments qui sont liés à l'élément ec_{ij} par une relation du modèle RBAC.

Le processus d'identification des contraintes des éléments communs consiste à parcourir le graphe de contraintes définies sur ces éléments.



L'exemple de parcours du graphe des contraintes pour identifier l'ensemble des contraintes de role commun r.

les symboles: r, f, p, m, o signifient: roles, fonctions, permissions, méthodes et objets
les lignes signifient les contraintes définies sur les éléments
c1, c2, c3, c4, ... − les contraintes d'un parcours

FIG. 4.13 – Un exemple de graphe de contraintes définies sur les éléments du modèle.

Supposons, que l'on cherche l'ensemble des contraintes définies pour un rôle commun $r_j \in R'$ présenté dans la Figure 4.13. Il contient l'ensemble des contraintes déjà définies pour ce rôle dans le système $C_S(r_j)$ et l'ensemble des contraintes définies dans l'application $C_A(r_j)$. On peut identifier l'ensemble des contraintes définies pour ce rôle dans le système et dans l'application, "directement" et "indirectement".

L'identification de ces contraintes demande le parcours du graphe de contraintes définies pour ce rôle et pour les éléments attachés à ce rôle directement et indirectement. Le parcours présenté dans la Figure 4.13 détermine un des ensembles de contraintes pour le rôle commun r_j, i.e. l'ensemble : $\{c_1, c_2, c_3, c_4, ..., c_9\}$.

Le parcours du graphe de contraintes construit les ensembles des contraintes définies pour un élément commun $ec_{ij} \in E'_i$ comme suit :

$$C_S(ec_{ij}) = c_S(ec_{ij}) \cup \bigcup C_S(el_{ij})$$

$$C_A(ec_{ij}) = c_A(ec_{ij}) \cup \bigcup C_A(el_{ij})$$

$c_S(ec_{ij})$ - *un ensemble de contraintes définies sur un élément* ec_{ij} *dans le système*
$c_A(ec_{ij})$ - *un ensemble de contraintes définies sur un élément* ec_{ij} *dans la nouvelle application*

$el_{ij} \in E_i$ - *un élément attaché à un élément* ec_{ij}, *par exemple pour* $ec_{ij} = f$ *(fonction) un élément* $el_{ij} = p$ *(permission) ou* $el_{ij} = r$ *(rôle)*
$C_S(el_{ij})$ - *un ensemble de contraintes définies sur un élément* el_{ij} *attaché directement ou indirectement à* ec_{ij} *dans les système*
$C_A(el_{ij})$ - *un ensemble de contraintes définies sur un élément* el_{ij} *attaché directement ou indirectement à* ec_{ij} *dans la nouvelle application*

En particulier les contraintes définies pour chacun des éléments communs sont les suivantes :

1. pour une méthode m_j :

$$C_S(m_j) = c_S(m_j) \cup \bigcup C_S(el_j) \quad et \quad C_A(m_j) = c_A(m_j) \cup \bigcup C_A(el_j)$$

où $C_S(el_j) ::= C_S(p_j)$ et $C_A(el_j) ::= C_A(p_j)$

$$C(m_j) = C_S(m_j) \cup C_A(m_j)$$

2. pour un objet o_j :

$$C_S(o_j) = c_S(o_j) \cup \bigcup C_S(el_j) \quad et \quad C_A(o_j) = c_A(o_j) \cup \bigcup C_A(el_j)$$

où $C_S(el_j) ::= C_S(p_j)$ et $C_A(el_j) ::= C_A(p_j)$

$$C(o_j) = C_S(o_j) \cup C_A(o_j)$$

3. pour une permission p_j :

$$C_S(p_j) = c_S(p_j) \cup \bigcup C_S(el_j) \quad et \quad C_A(p_j) = c_A(p_j) \cup \bigcup C_A(el_j)$$

où $C_S(el_j) ::= C_S(f_j) \mid C_S(m_j) \mid C_S(o_j)$ et $C_A(el_j) ::= C_A(f_j) \mid C_A(m_j) \mid C_A(o_j)$

$$C(p_j) = C_S(p_j) \cup C_A(p_j)$$

4. pour une fonction f_j :

$$C_S(f_j) = c_S(f_j) \cup \bigcup C_S(el_j) \quad et \quad C_A(f_j) = c_A(f_j) \cup \bigcup C_A(el_j)$$

où $C_S(el_j) ::= C_S(r_j) \mid C_S(p_j)$ et $C_A(el_j) ::= C_A(r_j) \mid C_A(p_j)$

$$C(f_j) = C_S(f_j) \cup C_A(f_j)$$

5. pour un rôle r_j :

$$C_S(r_j) = c_S(r_j) \cup \bigcup C_S(el_j) \quad et \quad C_A(r_j) = c_A(r_j) \cup \bigcup C_A(el_j)$$

où $C_S(el_j) ::= C_S(u_j) \mid C_S(f_j)$ et $C_A(el_j) ::= C_A(u_j) \mid C_A(f_j)$

$$C(r_j) = C_S(r_j) \cup C_A(r_j)$$

Nous avons identifié les ensembles des contraintes définies pour un élément commun ec_{ij} de l'ensemble E'. Ces ensembles limitent les éléments accessibles pour l'élément.

Les algorithmes 4.2 et 4.3 présentent l'étape d'identification des contraintes des éléments communs définies dans le système et/ou dans la nouvelle application.

Algorithme 4.2 L'identification des contraintes des éléments communs

$identifierContraintes(element_{ij})$

Début
  pour *chaque* $E_i \in E$ **faire**
    pour *chaque* $el_{ij} \in E_i$ **faire**
      $setMarque(el_{ij}, faux)$
    fait
  fait
  $EnsembleC(element_{ij}) = parcoursGrapheContraintes(element_{ij})$
  retourner $EnsembleC(element_{ij})$
Fin

Algorithme 4.3 Le parcours du graphe de contraintes pour trouver les contraintes définies

$parcoursGrapheContraintes(element_{ij})$

Début
  $setMarque(element_{ij}, vrai)$
  $CPGC(element_{ij}) = \emptyset$
  $CD(element_{ij}) = trouverContraintesDirect(element_{ij})$
  $EAt(element_{ij}) = trouverElementsAttaches(element_{ij})$
  pour *chaque* $el_{ij} \in EAt(element_{ij})$ **faire**
    si $(getMarque(el_{ij}) == faux)$ **alors**
      $CEAt(el_{ij}) = CEAt(el_{ij}) \cup parcoursGrapheContraintes(el_{ij})$
    finSi
  fait
  $CPGC(element_{ij}) = CD(element_{ij}) \cup CEAt(el_{ij})$
  retourner $CPGC(element_{ij})$
Fin

4.3.2.5 L'étape de vérification de la cohérence des éléments communs

Cette étape de l'algorithme vérifie que les éléments communs du système et de la nouvelle application sont cohérents, c'est-à-dire que la définition de la cohérence est toujours satisfaite après l'ajout de la nouvelle application. Donc, pour vérifier la cohérence des éléments communs, il faut :

- déterminer les éléments accessibles pour chaque élément commun, i.e. identifier un ensemble $E_S(ec_{ij})$ - ces éléments sont limités par les contraintes définies au niveau du système $C_S(ec_{ij})$ et au niveau de la nouvelle application $C_A(ec_{ij})$,

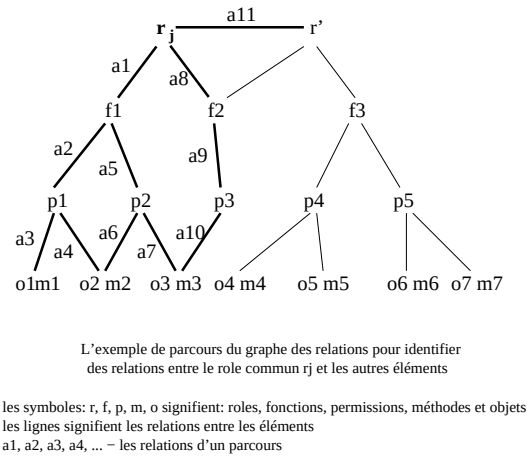


FIG. 4.14 – Un exemple de graphe de relations d'un élément commun avec les autres éléments.

– vérifier que la définition de la cohérence est toujours satisfaite pour chaque élément commun avec l'ensemble des éléments accessibles $E_S\,(ec_{ij})$.

L'identification des éléments accessibles pour chaque élément commun ec_{ij} doit être réalisée avec le parcours du graphe de relations (i.e. associations) de l'élément commun avec les autres éléments en accord avec les contraintes identifiées au niveau du système $C_S\,(ec_{ij})$ et au niveau de la nouvelle application $C_A\,(ec_{ij})$.

Supposons, que l'on cherche l'ensemble des éléments accessibles pour un rôle commun $r_j \in R'$ présenté dans la Figure 4.14. L'identification de ces éléments demande l'identification des relations entre ce rôle commun et les autres éléments. Le parcours présenté dans la Figure 4.14 détermine un des ensembles de relations du rôle commun r_j, i.e. l'ensemble : $\{a_1, a_2, a_3, a_4, ..., a_{11}\}$, en tenant compte des contraintes identifiées pour ce rôle commun.

Les algorithmes 4.4 et 4.5 présentent l'étape de vérification de la cohérence des éléments communs.

Algorithme 4.4 La vérification de la cohérence des éléments communs

$verifierCoherence\,(element_{ij},\ C_S\,(element_{ij})\ ,\ C_A\,(element_{ij}))$

Début
  pour *chaque* $E_i \in E$ **faire**
    pour *chaque* $el_{ij} \in E_i$ **faire**
      $setMarque\,(el_{ij}, faux)$
    fait
  fait
  $E_S\,(element_{ij}) = parcoursGrapheRelations\,(element_{ij},\ C_S\,(element_{ij})\ ,\ C_A\,(element_{ij}))$
  pour *chaque* $E_{Sk}\,(element_{ij}) \in E_S\,(element_{ij})$ **faire**
    si $(E_{Sk}\,(element_{ij}) == \emptyset)$ **alors**
      retourner faux
    finSi
  fait
  retourner vrai
Fin

Si la définition de la cohérence est satisfaite pour chaque élément commun, alors le système après l'ajout de la nouvelle application est encore cohérent. Dans le cas contraire, il faut éliminer

Algorithme 4.5 Le parcours du graphe de relations

$parcoursGrapheRelations\,(element_{ij},\ C_S\,(element_{ij})\ ,\ C_A\,(element_{ij}))$

Début
$setMarque\,(element_{ij}, vrai)$
$EAccess\,(element_{ij}) = \emptyset$
$EAccess\,(element_{ij}) = trouverElementsAccessibles\,(element_{ij})$
$EAccess\,(element_{ij}) = EAccess\,(element_{ij}) \cap$
$verificationElementsAvecContraintes\,(element_{ij},\ C_S\,(element_{ij})\ ,\ C_A\,(element_{ij}))$
pour *chaque* $el_{ij} \in EAccess\,(element_{ij})$ **faire**
si $(getMarque\,(el_{ij}) == faux)$ **alors**
$C_S\,(el_{ij}) = identifierContraintes\,(el_{ij})$
$C_A\,(el_{ij}) = identifierContraintes\,(el_{ij})$
$EAccessIndirect\,(element_{ij}) =$
$EAccessIndirect\,(element_{ij}) \cup parcoursGrapheRelations\,(el_{ij},\ C_S\,(el_{ij})\ ,\ C_A\,(el_{ij}))$
finSi
fait
$EAccess\,(element_{ij}) = EAccess\,(element_{ij}) \cup EAccessIndirect\,(element_{ij})$
retourner $EAccess\,(element_{ij})$
Fin

les incohérences en mettant en place des changements dans les contraintes. Il faut donc identifier les contraintes qui sont responsables des incohérences éventuelles du système.

4.3.2.6 L'étape de recherche des contraintes responsables

Lorsque l'on aboutit à un système incohérent à la suite de l'ajout d'une nouvelle application, il peut être utile de fournir à l'administrateur la liste des contraintes qui ont conduit à l'incohérence globale.

Il est nécessaire de vérifier quelles contraintes des ensembles $C_S\,(ec_{ij})$ et $C_A\,(ec_{ij})$ provoquent des situations dans lesquelles la définition de la cohérence n'est pas satisfaite :

pour chaque élément commun $ec_{ij} \in E'_i$ *pour lequel des incohérences sont identifiées, trouver les ensembles des contraintes qui sont responsables de ces incohérences, i.e. ensembles qui appartient aux ensembles* $C_S\,(ec_{ij})$ *et* $C_A\,(ec_{ij})$.

L'identification des contraintes qui provoquent les incohérences consiste à identifier pour $ec_{ij} \in E'_i$ les ensembles $E_{Sk}\,(ec_{ij}) \in E_S\,(ec_{ij})$ tels que : $E_{Sk}\,(ec_{ij}) = \emptyset$.

L'algorithme 4.6 présente l'étape de la recherche des contraintes qui provoquent des incohérences pour un élément commun ec_{ij}.

Le choix de la ou des contraintes qu'il faudra éliminer ou modifier doit être fait par l'administrateur de sécurité et/ou par le concepteur, car il peut exister de nombreuses solutions.

On peut noter néanmoins, que les modifications faites par l'administrateur auront un impact sur l'ensemble du système, alors que les modifications au niveau de l'application seront plus localisées.

Les modifications des contraintes au niveau de l'administrateur pourront provoquer des changement indésirables dans le système, dans ce cas la meilleure solution est la réalisation de modifications au niveau du concepteur de la nouvelle application.

L'administrateur de sécurité doit choisir le niveau auquel les modifications de contraintes peuvent être réalisées en prenant en considération :
- la simplicité et la vitesse des modifications,
- la facilité de la vérification de cohérence du système,
- les profits futurs au niveau du système global.

Algorithme 4.6 L'identification des contraintes responsables d'incohérence d'un élément commun

$trouverContraintesIncoherentes\,(element_{ij},\ C_S\,(element_{ij})\ ,\ C_A\,(element_{ij}))$

Début
  pour ($chaque\ c_S \in C_S\,(element_{ij}) \wedge chaque\ c_A \in C_A\,(element_{ij})$) **faire**
    $EAccess_S\,(element_{ij}) = parcoursGrapheRelations\,(element_{ij},\ c_S,\ c_A)$
    pour $chaque\ EAccess_{Sk}\,(element_{ij}) \in EAccess_S\,(element_{ij})$ **faire**
      si ($EAccess_{Sk}\,(element_{ij}) = \emptyset$) **alors**
        $CI_S\,(element_{ij}) = CI_S\,(element_{ij}) \cup c_S$
        $CI_A\,(element_{ij}) = CI_A\,(element_{ij}) \cup c_A$
      finSi
    fait
  $CI\,(element_{ij}) = CI_S\,(element_{ij}) \cup CI_A\,(element_{ij})$
  retourner $CI\,(element_{ij})$
Fin

Après modifications, les étapes de vérification de la cohérence doivent être réexécutées pour aboutir en définitif à un nouveau système cohérent.

4.4 Conclusion

Les contraintes sont un aspect important du contrôle d'accès pour organiser la politique de sécurité au plus haut niveau. Elles sont nécessaires dans un système d'information pour garantir une cohérence globale dans les règles d'administration du système d'information, du coté du concepteur et du coté de l'administrateur. Elles permettent aussi de mettre en place la séparation de compétence dans le système d'information.

Nous avons divisé, en général, les contraintes en deux groupes, les contraintes au niveau du concepteur et les contraintes au niveau de l'administrateur de sécurité. Le concepteur d'une application n'a pas la vision de la politique globale de sécurité, d'un autre coté l'administrateur n'a pas une connaissance fine de l'application qu'il doit introduire dans le système global.

Nous avons montré que ces contraintes peuvent être définies aux différents niveaux en utilisant des moyens d'expression différents. Donc, la traduction de ces contraintes est une étape très importante de la validation de cohérence du système. La traduction doit être réalisée pour tous les éléments du modèle et pour toutes les contraintes qui sont définies, dans notre approche, au niveau du concepteur de l'application.

L'étape de validation consiste en une vérification de la cohérence globale du système. Cette cohérence peut être enfreinte par l'ajout de nouveaux éléments, i.e. nouvelles contraintes, nouvelles applications, etc., dans le système. Donc, il faut vérifier la cohérence du système après chaque modification réalisée dans ce système. Une vérification qui retourne un résultat positif, garantit la cohérence globale du système et permet son utilisation.

Les concepts décrits dans le Chapitre 3 sont implémentés sur la plate-forme présentée dans le chapitre suivant. Les aspects contrainte décrits dans ce chapitre sont en cours d'implémentation.

Chapitre 5

Une plate-forme pour la production de rôles

Objectifs du chapitre :
- *Présenter les outils de type AGL (CASE)*
- *Introduire les langages de description XML et XMI*
- *Présenter la plate-forme pour la production de rôles d'un système d'information*
  - *Présenter l'algorithme de création de rôles*
  - *Décrire le document résultat*
  - *Développer l'outil de l'administrateur de sécurité*

Nous proposons dans ce chapitre un outil qui permettra à l'administrateur de sécurité de gérer la sécurité du système d'information au niveau du contrôle d'accès.

On propose une plate-forme qui va mettre en oeuvre les concepts présentés dans les Chapitre 3 et Chapitre 4 (Figure 5.1). Son objectif est la coopération entre le concepteur et l'administrateur de sécurité pour réaliser et intégrer les concepts du contrôle d'accès présentés dans le Chapitre 2. Le résultat final de cette plate-forme est la validation des activés du concepteur et de l'administrateur qui garantit la cohérence globale au niveau du contrôle d'accès du système d'information.

Nous avons choisi différents outils pour réaliser la plate-forme, ils permettent l'indépendance avec l'environnement cible :
- un logiciel de type AGL qui se base sur les technologies objet et qui permet d'utiliser les concepts d'UML,
- le langage d'échange de données XML [Mic99, XML] qui garantit un formalisme indépendant,
- le langage de programmation Java [Jav] qui est orienté objet et indépendant du système cible.

L'objectif de ce chapitre est la présentation d'une plate-forme UML-XML-Java de production de rôles. Le chapitre est divisé en trois parties. La première partie présente les outils qui ont été utilisés pour la réalisation de cette plate-forme, les outils de type AGL (CASE) et le langage de description XML et XMI. La deuxième partie décrit notre plate-forme qui permet l'ingénierie de rôles en se basant sur le modèle RBAC étendu. La troisième présente un outil qui aide l'administrateur de sécurité à réaliser les tâches décrites dans le Chapitre 4.

5.1 Les outils utilisés pour la plate-forme

Nous avons choisi un logiciel de type AGL pour utiliser les concepts du langage UML. Les langages XML et XMI sont utilisés pour échanger les informations qui viennent de différents

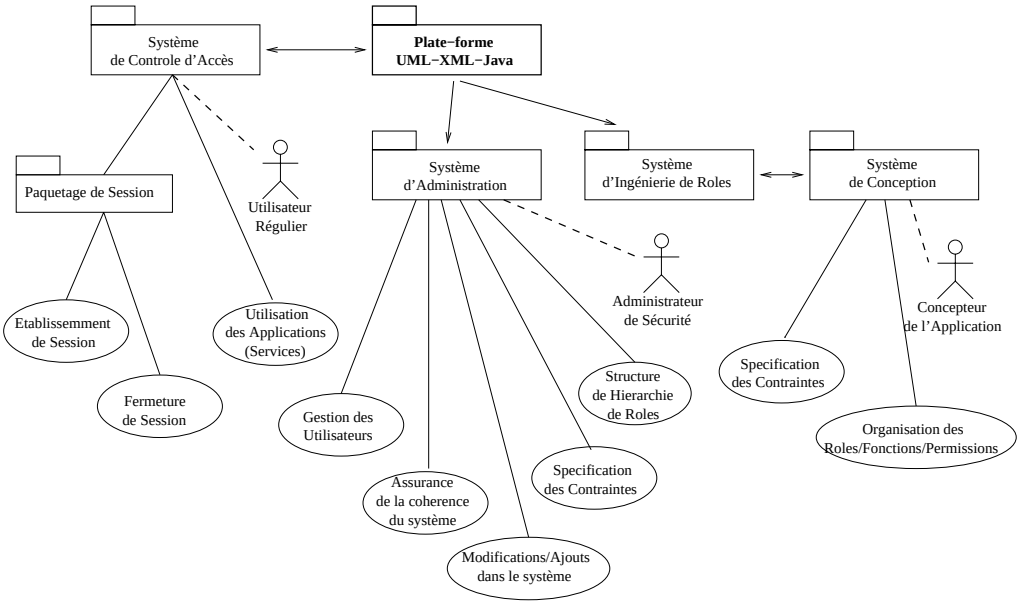


FIG. 5.1 – La fonctionnalité de la plate-forme UML-XML-Java.

diagrammes d'UML.

5.1.1 Les logiciels utilisant le langage UML

Aujourd'hui de nombreux outils AGL sont disponibles sur le marché. Parmi ceux-ci de nombreux utilisent une méthode de conception orientée objet associée au langage UML.

AGL - *Atelier Génie Logiciel* (***CASE*** - *Computer Aided Software Engineering* ou *Computer Aided System Engineering*) - un système qui aide l'analyse et la conception des systèmes d'information en se basant sur la technologie objet. L'AGL contient en général :

- une interface graphique pour désigner, modifier et imprimer les différents diagrammes, e.g. les diagrammes d'UML,
- des procédures pour générer automatiquement des schémas de bases de données relationnelles (e.g. dans SQL) ou des schémas objets (e.g. dans IDL),
- une vérification de cohérence formelle entre les différents diagrammes,
- un dictionnaire commun des noms utilisés et une vérification de cohérence de leur utilisation,
- une spécification des fonctions applicatives réalisées dans un système ou une base de données projetée par la définition des données d'entrée, des données de sortie,
- une aide à la création de documentation pour les différents phases de la conception et pour les différents niveaux d'abstraction,
- une création automatique du prototype de l'application en se basant sur une spécification formalisée.

Au cours de nos travaux, nous avons utilisé les AGL suivants : ***Rose*** de Rational Inc. [Rat] et ***Argouml*** de University of California, Berkeley [Arg].

5.1.2 Le langage XML

XML (***eXtensible Markup Language***) est un langage de description et d'échange de documents structurés. Il permet de décrire la structure logique de documents textuels, à l'aide d'un système de balises permettant de marquer les éléments qui composent la structure et les relations entre ces éléments [Mic99]. XML apparaît comme un format d'échange universel de documents créés à l'aide de systèmes de traitement de documents.

XML est le résultat de la coopération d'un grand nombre d'entreprises et de chercheurs partenaires du World Wide Web Consortium (W3C) [Gro00b, XML]. XML a été créé pour améliorer les fonctionnalités du Web en fournissant un moyen flexible d'échange des informations. Comme HTML, il est basé sur l'utilisation de balises mais contrairement à lui XML n'est pas basé sur un format fixe. Son objectif est de définir un formalisme permettant d'échanger facilement des documents complexes sur le Web, en dépassant les limites imposées par HTML.

Les avantages décisifs que XML apporte au niveau applicatif sont :

1. *Extensibilité et structure* - possibilité de définir des éléments spécifiques pour chaque composant identifié d'une information structurée. Il permet d'organiser ces éléments dans une structure complexe et d'exploiter cette organisation pour échanger des données dans les domaines d'application les plus divers.
2. *Modularité et réutilisation des structures types* - tout utilisateur de XML peut définir librement sa propre structure de document et les autres utilisateurs seront parfaitement capables de visualiser convenablement les documents d'autrui.
3. *Contrôle de validité* - un document XML se conformant à une structure type explicite pourra être valide. La vérification de conformité peut être lexicale, structurelle, syntaxique et sémantique.
4. *Accès à des sources d'information hétérogènes* - il est très fréquent qu'une groupe d'utilisateurs exploite des bases de données hétérogènes, les technologies hétérogènes portant aussi bien sur les technologies de SGBD (relationnel, systèmes documentaires, systèmes de représentation des connaissances, etc.) que sur les structures de données. XML contribue à résoudre ce problème en proposant un format d'échange de données normalisé, général, indépendant de toute plate-forme.

Un extrait de la syntaxe de langage XML est présenté dans l'Annexe D.

5.1.3 Le langage XMI

XMI (***XML Metadata Interchange***) est une utilisation de XML qui facilite les échanges d'informations de haut niveaux de type méta-donnée (description d'une donnée et de la façon dont elle est organisée) [XMI]. De façon plus spécifique, XMI permet aux utilisateurs d'UML d'échanger leur modèle de façon indépendante de l'AGL utilisé.

XMI est une proposition de l'OMG bâtie sur les trois standards suivants :
- XML définit par le W3C,
- UML définit par l'OMG,
- MOF (Meta Object Facility) également définit par l'OMG.

XMI est le format standard de sérialisation de modèles défini par l'OMG en accord avec le W3C. Ce format permet d'échanger toute sorte de modèles basés sur le MOF et pas seulement des modèles UML. XMI est basé sur XML, sur le MOF et sur OCL.

XMI utilisé avec UML et MOF forment le noyau de l'architecture d'OMG qui intègre des outils de modélisation et conception orientée objet.

5.2 La production des rôles d'un modèle RBAC étendu

Le processus de production des rôles basé sur le modèle RBAC étendu utilise le langage UML, le langage XML et XMI. Une application a été écrite en Java [Jav]. La Figure 5.2 présente les étapes de la production des rôles.

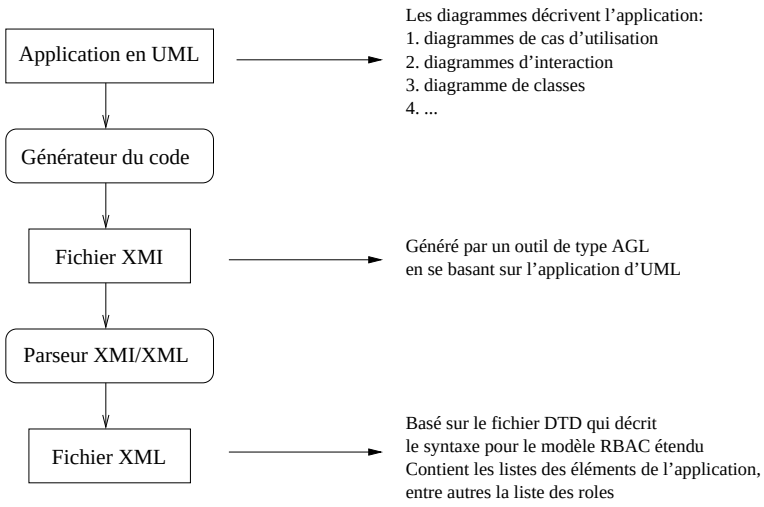


FIG. 5.2 – Le processus de production des rôles d'un modèle RBAC.

5.2.1 La création des rôles - la définition d'un ensemble de rôles

Nous avons présenté dans le Chapitre 3 le rapprochement des concepts du modèle RBAC étendu et d'UML. On a proposé une spécification du contrôle d'accès de sécurité au niveau du schéma conceptuel du système d'information en utilisant le langage de modélisation UML.

Nous allons spécifier l'algorithme de construction des rôles du système d'information à partir de sa représentation en utilisant le méta-modèle d'UML.

5.2.1.1 La présentation de l'algorithme pour construire les rôles

Les étapes de cet algorithme sont les suivantes (Algorithme 5.1) :
- associer un ensemble de permissions (couples (*méthode*, *objet*)) à une fonction (cas d'utilisation du modèle de conception),
- associer un ensemble de fonctions à un rôle (acteur dans le modèle de conception).

Dans la première étape nous allons déterminer l'ensemble des fonctions qui sera déduit du parcours des différents diagrammes de cas d'utilisation qui composent le modèle de l'application. Ensuite, on trouve les permissions par les parcours des diagrammes de séquences correspondants aux cas d'utilisation représentants les fonctions. On construit les ensembles de permissions et on les associe aux fonctions. La deuxième étape consiste à identifier les rôles du modèle et associer les fonctions à ces rôles.

Pour cela, nous allons utiliser la partie du méta-modèle d'UML qui décrit les diagrammes de cas d'utilisation et les diagrammes de séquence.

Algorithme 5.1 La construction des rôles

ConstructionRoles(Modele)

Début
AffectationDesPermissionsAuxFonctions(Modele)
AffectationDesFonctionsAuxRoles(Modele)
Fin

5.2.1.2 La première étape de l'algorithme - affectation des permissions aux fonctions

La première étape de l'algorithme ***Affectation Des Permissions Aux Fonctions*** recherche les fonctions et les permissions du modèle. Elle construit l'ensemble des permissions pour chaque fonction (i.e. pour chaque cas d'utilisation du diagramme de cas d'utilisation).

La recherche des ensembles de permissions *ensemblePermissions* associés aux fonctions du modèle est donnée par la fonction **AffectationDesPermissionsAuxFonctions(Modele)** (Algorithme 5.2)

Algorithme 5.2 La première étape de l'algorithme de la création des rôles

AffectationDesPermissionsAuxFonctions(Modele : Model)

Début
  $ensembleCU = RechercherCU\,(Modele)$
  pour *chaque* $cu_i \in ensembleCU$ **faire**
    $ensemblePermissions = CreationEnsemblePermissions\,(cu_i)$
  fait
Fin

La première partie commence par la recherche de l'ensemble des cas d'utilisation du diagramme de cas d'utilisation et l'affecte à *ensembleCU*. Pour cela on utilise la méthode ***RechercherCU(Modele)*** - Algorithme 5.3.

Cette méthode permet de retrouver tous les cas d'utilisation du modèle présentés dans le diagramme de cas d'utilisation et renvoie l'ensemble des cas d'utilisation.

Dans l'application "Gestion des Notes" le diagramme de cas d'utilisation possède l'ensemble des cas d'utilisation suivant :

ensembleCasUtilisation = (Visualisation, Edition, Saisie les notes, Configuration, Validation d'utilisateur, Visualiser la liste complète, Visualiser le bulletin, Visualiser les notes, Edition de la liste complet, Edition du bulletin)

Nous avons indiqué dans le Chapitre 3 qu'une fonction de notre modèle RBAC étendu correspond à un cas d'utilisation du langage UML. L'identification des fonctions consiste à rechercher les cas d'utilisation qui composent le modèle de l'application. On peut effectuer un parcours du méta-modèle d'UML pour retrouver les cas d'utilisation ainsi que les relations qui existent entre eux (extension, utilisation). La partie du méta-modèle qui décrit les cas d'utilisation se retrouve dans le paquetage *Model Management* (Figure de l'extrait du paquetage ModelManagement, Annexe B).

Dans le méta-modèle d'UML, un modèle est une abstraction du système d'information. La relation d'héritage qui se trouve entre *Model* et *Package* exprime qu'un *Model* est une sous-classe de *Package* et peut contenir des éléments de modélisation (via la relation d'agrégation entre *Package* et *ModelElement*) ou des paquetages. Un paquetage regroupe un ensemble d'éléments de modélisation. Un cas d'utilisation est un élément de modélisation et la recherche des cas d'utilisation peut être faite à partir du modèle *Model* qui les contient.

Par le paquetage *Use_ Cases* (du paquetage *Behavioral Elements*) on remarque qu'un *cas d'utilisation* est une sous-classe de *Classifier* via la relation d'héritage entre le *Classifier* et *Use Case* et la relation d'héritage entre le *Classifier* et *Actor* - Figure de l'extrait du paquetage Use_Cases (de Behavioral Elements), Annexe B.

D'autre part *Classifier* est la sous-classe de *ModelElement* dans le paquetage *Foundation Core - Backbone* parce que *Classifier* est une sous-classe de *GeneralizableElement* et une sous-classe de

NameSpace (les sous-classes de *ModelElement*) - Figure de l'extrait du paquetage Backbone (de Core), Annexe B.

On s'intéresse aux paquetages de la vue des cas d'utilisation. Chaque cas d'utilisation est ajouté à l'ensemble des cas d'utilisation.

Algorithme 5.3 La méthode *RechercherCU(Modele)*

RechercherCU(Modele : Model)

Début
  pour *chaque* $modele_i \in Modele$ **faire**
   si ($modele_i$ *de type* $UseCaseView$) **alors**
    pour *chaque* $modelElement_j \in modele_i$ **faire**
     si ($modelElement_j$ *de type* $UseCase$) **alors**
      $ensemble\_CU = ensemble\_CU \cup modelElement_j$
     finSi
     si ($modelElement_j$ *de type* $Package$) **alors**
      $ensemble\_CU = ensemble\_CU \cup RechercherCU\ (modelElement_j)$
     finSi
    fait
   finSi
  fait
  retourner $ensemble\_CU$
Fin

Pour chaque cas d'utilisation identifié dans le modèle, nous allons associer l'ensemble des permissions nécessaires à son exécution - la méthode ***CreationEnsemblePermissions(casUtilisation)*** - Algorithme 5.4.

Cette méthode permet de trouver les permissions d'un cas d'utilisation par la construction et renvoie un ensemble de permissions.

Le comportement d'un cas d'utilisation est décrit au moyen de *collaborations* entre les objets,, en accord avec le paquetage *Behavioral Elements - Collaboration.*

Un cas d'utilisation est un *Classifier* et il est décrit par une ou plusieurs collaborations à partir de l'association entre le *Classifier* et la *Collaboration.* La multiplicité de l'association indique qu'il n'existe qu'une collaboration pour un *Classifier.* Alors, il existe toujours une collaboration du cas d'utilisation et on peut établir son ensemble de permissions - Figure de l'extrait du paquetage Collaborations - Roles (de Behavioral Elements) et Figure de l'extrait du paquetage Collaborations - Interactions (de Behavioral Elements), Annexe B.

Par exemple le cas d'utilisation *Configuration* de l'application "Gestion des Notes" se compose de collaborations suivantes : *Saisie d'un Enseignant*, *Saisie d'un Etudiant*, *Saisie d'une Matière*, *Affectation d'un Enseignant à une Matière* et *Affectation d'un Etudiant à une Matière.*

L'association entre *Classifier* et *Collaboration* nous permettra d'obtenir ces collaborations.

On veut trouver les permissions de la collaboration *Saisie d'un Enseignant* du cas d'utilisation *Configuration.* Par définition une collaboration peut avoir plusieurs interactions. Par exemple, dans le diagramme de séquence *Saisie d'un Enseignant* on a les méthodes :

- création d'un nouveau Enseignant - *création()*,
- modification des informations d'un Enseignant - *modification().*

Dans le méta-modèle, le passage d'une collaboration à ses interactions est fait par l'agrégation entre *Collaboration* et *Interaction.* On peut accéder aux messages de l'interaction. Une interaction est la séquence de *messages* spécifiant la communication entre l'ensemble des instances. Il

Algorithme 5.4 La méthode *CreationEnsemblePermissions(casUtilisation)*

CreationEnsemblePermissions(cu : casUtilisation)

Début
  pour *chaque collaboration* $\subset cu$ **faire**
    pour *chaque interaction* **faire**
      pour *chaque message* **faire**
        $permission = RechercherElementsPermission\,(message)$
        si $(permission \notin ensemblePermissions)$ **alors**
          $ensemblePermissions_{cu} = ensemblePermissions_{cu} \cup permission$
        finSi
      fait
    fait
  fait
  retourner $ensemblePermissions_{cu}$
Fin

y a une relation d'agrégation entre *Interaction* et *Message* - Figure de l'extrait du paquetage Collaborations - Interactions (de Behavioral Elements), Annexe B.

Par exemple, l'interaction *Création()* de l'application "Gestion des Notes" possède la séquence de messages suivante :

{création(), active(), ajouterEnseignant(), newEnseignant(nom, prenom, nomlogin, motdepasse), ajouter(Enseignant), ajouter(Enseignant), cache()}

Un *message* définit une utilisation d'une *interaction*. Il spécifie les rôles d'expéditeur et du récepteur en même temps l'envoi d'une *action* - le passage de la *CallAction*.

Par la multiplicité de l'association plusieurs messages peuvent être associés à la même *Action*. Par exemple, on cherche les permissions associées au message *newEnseignant(nom, prenom, nomlogin, motdepasse)*. Le type de l'action est *CallAction* - Figure de l'extrait du paquetage Common Bahavior - Actions (de Behavioral Elements), Annexe B.

On peut obtenir une opération par l'association entre *CallAction* et *Operation*. Une *CallAction* est une sous-classe de *Action* dans le méta-modèle d'UML.

Pour obtenir le nom de la méthode, le nom de l'objet et la contrainte associée on utilise la fonction ***RechercherElementsPermission(message)*** - Algorithme 5.5.

Algorithme 5.5 La méthode *RechercherElementsPermission(message)*

RechercherElementsPermission(message : Message)

Début
  $operation = RechercherOperation\,(message)$
  $m_i = RechercherNomMethode\,(operation)$
  $o_j = RechercherNomObjet\,(operation)$
  $c_k = RechercherContrainte\,(m_i, o_j)$
  $permission = CreationPermission\,(m_i, o_j, c_k)$
  retourner *permission*
Fin

m_i est le nom de la méthode,
o_j est le nom de l'objet de la méthode,

c_k est l'ensemble des contraintes.

La méthode ***RechercherOperation(message)*** donne une opération de la méthode par le rôle *action* de l'association - elle a comme argument le *Message* et renvoie l'*Operation* associée.

La méthode ***RechercherNomMethode(operation)*** retourne le nom de la méthode - par un attribut *name* hérité de *ModelElement* (Algorithme 5.6).

Algorithme 5.6 La méthode *RechercherNomMethode(operation)*

RechercherNomMethode(operation : Operation)

Début
retourner operation.name
Fin

La méthode ***RechercherNomObjet(operation)*** retourne le nom de l'objet de la méthode que l'on veut exécuter - par une association entre le *Feature* et le *Classifier* (le paquetage *Foundation Core - Backbone* - Figure de l'extrait du paquetage Backbone de Core, Annexe B). Le rôle qu'on utilise est *owner*. On peut utiliser cette association parce que une *Operation* et une sous-classe de *Behavioral Feature* et *Behavioral Feature* est une sous-classe de *Feature* - une *Operation* peut utiliser une association décrite pour le *Feature* (Algorithme 5.7).

Algorithme 5.7 La méthode *RechercherNomObjet(operation)*

RechercherNomObjet(operation : Operation)

Début
retourner operation.owner.name
Fin

La méthode ***RechercherContrainte(methode, objet)*** - Algorithme 5.8 - donne l'ensemble des contraintes associées à la méthode. S'il existe des contraintes associées à l'objet alors cela veut dire que ces contraintes sont valables pour toutes les opérations de cet objet et on les ajoute à l'ensemble des contraintes.

On présente l'exemple de contrainte du diagramme de séquence *Saisir les Notes* : pour réaliser le cas d'utilisation, un *Enseignant* doit créer un *contrôle* - un *Contrôle* peut être créé seulement par un *Enseignant* responsable de la *Matière*. On prend les éléments suivants du diagramme de cas d'utilisation et du diagramme de séquence :

- cas d'utilisation : *Saisir les Notes*,
- collaboration : *Saisie de Note*,
- interactions : *Création()*, *Modification()*,
- par exemple, pour *création() :*
  - message : par exemple *créationContrôle(Matière)*
  - opération :
    - le nom de la méthode : *créationContrôle*
    - le nom de l'objet (le nom de la classe) : *contrôle (Contrôle)*
    - la contrainte : *seulement un Enseignant de la Matière peut exécuter cette méthode.*

Ces trois méthodes, ***RechercherNomMethode(operation)***, ***RechercherNomObjet(operation)*** et ***RechercherContrainte(methode, objet)*** permettent de construire les permissions. Ensuite,

Algorithme 5.8 La méthode *RechercherContrainte(methode, objet)*

RechercherContrainte ($m_i : Method$, $o_j : Object$)

Début
  si (o_j possède les contraintes associées) **alors**
    pour *chaque* $contrainte_l$ **faire**
      $c_k = c_k \cup contrainte_i$
    fait
  finSi
  si (m_i possède les contraintes associées) **alors**
    pour *chaque* $contrainte_j$ **faire**
      $c_k = c_k \cup contrainte_j$
    fait
  finSi
  retourner c_k
Fin

la méthode ***CreationPermission(methode, objet, contrainte)*** renvoie la permission de la méthode associée au message.

La méthode ***CreationEnsemblePermissions(casUtilisation)*** vérifie si cette permission existe déjà dans l'ensemble des permissions. Si elle n'existe pas, elle l'ajoute à l'ensemble $ensemblePermissions$. Toutes ces opérations sont faites pour chaque collaboration, pour chaque interaction et pour chaque message afin d'obtenir un ensemble de permissions associées au cas d'utilisation.

La méthode ***CreationEnsemblePermissions(casUtilisation)*** renvoie ensuite cet ensemble à la méthode ***AffectationDesPermissionsAuxFonctions(Modele)***. Une fois l'ensemble des permissions construit pour chaque cas d'utilisation du modèle, il faut compléter ceux-ci en fonction des relations entre les cas d'utilisation : relation d'utilisation et relation d'extension. On les traite dans la deuxième étape de l'algorithme.

La première étape de l'algorithme se termine par la création des ensembles des permissions associées aux fonctions du modèle.

Ayant une fonction qui définit l'ensemble des permissions associées au cas d'utilisation, il faut associer les fonctions aux rôles pour pouvoir construire les profils d'utilisateurs.

5.2.1.3 La deuxième étape de l'algorithme - affectation des fonctions aux rôles

La deuxième étape de l'algorithme consiste en ***Affectation Des Fonctions Aux Rôles***. Une fonction est l'ensemble des permissions associées au cas d'utilisation. L'association des fonctions aux rôles du modèle est donnée par la fonction **AffectationDesFonctionsAuxRoles(Modele)** (Algorithme 5.9)

où
$ensembleA$ est un ensemble d'acteurs,
$ensembleFA_{a_i}$ est un ensemble de fonctions d'un acteur a_i,
$ensembleRelationCU_{cu_j}$ est un ensemble de cas d'utilisation avec lesquels l'acteur est en relation par rapport à la relation d'utilisation ou la relation d'extension de cas d'utilisation cu_j,
$RechercherRelationCU\,(cu_j)$ est une fonction qui renvoie les cas d'utilisation avec lesquels un cas d'utilisation est en relation d'utilisation ou en relation d'extension,
$heritageEnsembleA_{a_i}$ est un ensemble d'acteurs avec lesquels l'acteur a_i est en relation d'héritage.

Algorithme 5.9 La deuxième étape de l'algorithme de la création des rôles

AffectationDesFonctionsAuxRoles(Modele : Model)

Début
$ensembleA = RechercherA\,(Modele)$
pour *chaque* $a_i \in ensembleA$ **faire**
$ensembleFA_{a_i} = CreationEnsembleFonctions\,(a_i)$
pour *chaque* $cu_j \in ensembleFA_{a_i}$ **faire**
$ensembleRelationCU_{cu_j} = RechercherRelationCU\,(cu_j)$
pour *chaque* $cu_k \in ensembleRelationCU_{cu_j}$ **faire**
si $(relation\,(cu_j, cu_k)$ *de type* $\ll uses \gg)$ **alors**
$ensemblePermissions_{cu_j} = ensemblePermissions_{cu_j} \cup ensemblePermissions_{cu_k}$
finSi
si $(relation\,(cu_j, cu_k)$ *de type* $\ll extends \gg)$ **alors**
$ensembleFA_{a_i} = ensembleFA_{a_i} \cup cu_k$
finSi
fait
fait
$heritageEnsembleA_{a_i} = RechercherRelationA\,(a_i)$
pour *chaque* $a_j \in heritageEnsembleA_{a_i}$ **faire**
$ensembleFA_{a_i} = ensembleFA_{a_i} \cup ensembleFA_{a_j}$
fait
fait
Fin

La deuxième étape de l'algorithme commence par la recherche de l'ensemble des acteurs du diagramme de cas d'utilisation et l'attribue à ***ensembleA*** - la méthode ***RechercherA (Modele)*** (Algorithme 5.10).

Cette méthode est réalisée de la même façon que la méthode *RechercherCU(Modele)* mais maintenant on cherche les éléments de modélisation de type d'acteur. Elle donne l'ensemble des acteurs du modèle décrits dans les diagrammes de cas d'utilisation.

Par exemple dans l'application "Gestion des Notes" l'ensemble des acteurs comme suit :

ensembleA = (Enseignant, Etudiant, Directeurs des Etudes, Secrétariat).

On peut créer pour chaque acteur du modèle, obtenu de cet ensemble *ensembleA*, l'ensemble des fonctions qu'il possède en interagissant avec le système - la méthode ***CreationEnsemble-Fonctions(acteur)*** - Algorithme 5.11. Cette méthode crée l'ensemble des fonctions d'un acteur dans le modèle.

Cette méthode prend comme argument un *acteur* qui est un *Classifier* (le paquetage *Behavioral Elements - Use Cases*) - Figure de l'extrait du paquetage Use_Cases (de Behavioral Elements), Annexe B. Un *Classifier* peut avoir plusieurs *associationEnd* donc un *acteur* peut interagir avec le système en étant en relations avec plusieurs cas d'utilisation ou acteurs.

On cherche pour chaque *associationEnd* de l'acteur une autre *associationEnd* qui est liée par la même *association* - pour trouver les cas d'utilisation pour un acteur. Il faut passer par l'association - elle possède au moins deux *associationEnd* ordonnées. Pour chaque *associationEnd* de l'association en la traversant par le rôle de connexion, on cherche si cette *associationEnd* est liée avec un *Classifier* de type *UseCase* - on peut récupérer la fonction représentée par le cas d'utilisation et on peut l'ajouter à l'ensemble des fonctions d'un acteur.

On peut aussi créer l'ensemble des fonctions pour chaque acteur. Il faut néanmoins la compléter

Algorithme 5.10 La méthode *RechercherA(Modele)*

RechercherA(Modele : Model)

Début
  pour *chaque* $modele_i \in Modele$ **faire**
    si ($modele_i$ *de type* $UseCaseView$) **alors**
      pour *chaque* $modelElement_j \in modele_i$ **faire**
        si ($modelElement_j$ *de type* $Actor$) **alors**
          $ensemble\_A = ensemble\_A \cup modelElement_j$
        finSi
        si ($modelElement_j$ *de type* $Package$) **alors**
          $ensemble\_A = ensemble\_A \cup RechercherA\ (modelElement_j)$
          finSi
      fait
    finSi
  fait
  retourner $ensemble\_A$
Fin

Algorithme 5.11 La méthode *CreationEnsembleFonctions(acteur)*

CreationEnsembleFonctions(a : Acteur)

Début
  pour *chaque* $associationEnd_i \subset a$ **faire**
    $association = Rechercher\ (associationEnd_i)$
    pour *chaque* $associationEnd_j \subset association$ **faire**
      $classifier = associationEnd_j.type$
      si ($classifier$ *de type* $UseCase$) **alors**
        $ensembleFonctionA = ensembleFonctionA \cup classifier$
      finSi
    fait
  fait
  retourner $ensembleFonctionA$
Fin

par l'ensemble des fonctions en relation avec le cas d'utilisation. Pour chaque cas d'utilisation de l'ensemble ***ensembleCU*** on cherche les cas d'utilisation avec lesquels il est en relation indirecte. Cet ensemble contient les cas d'utilisation liés par la relation <<uses>> ou par la relation <<extends>> - la méthode ***RechercherRelationCU(casUtilisation)*** (Algorithme 5.12).

Algorithme 5.12 La méthode *RechercherRelationCU(casUtilisation)*

RechercherRelationCU(cu : casUtilisation)

Début
  pour *chaque* $associationEnd_i \subset cu$ **faire**
    $association = Rechercher\,(associationEnd_i)$
    pour *chaque* $associationEnd_j \subset association$ **faire**
      $classifier = associationEnd_j.type$
      si ($classifier$ *de type* $UseCase$) **alors**
        $ensembleGeneralization = classifier.specialization$
        pour *chaque* $elementEnsemble_k \subset ensembleGeneralization$ **faire**
          si ($elementEnsemble_k.child$ *de type* $UseCase$) **alors**
            $ensembleRelationCU = ensembleRelationCU \cup elementEnsemble_k.child.specialization$
          finSi
        fait
      finSi
    fait
  fait
  retourner *ensembleRelationCU*
Fin

ensembleRelationCU est un ensemble de cas d'utilisation avec lesquels un cas d'utilisation est en relation.

Il existe une relation de communication entre un acteur et un cas d'utilisation - une *association* que l'on peut parcourir dans le méta-modèle d'UML en utilisant la même démarche que pour la méthode ***RechercherRelation(casUtilisation)***.

On peut créer l'ensemble des rôles pour chaque utilisateur. Il faut compléter un ensemble de rôles par les rôles qui viennent des relations d'héritage entre les acteurs. Pour chaque acteur de l'ensemble ***ensembleA*** on cherche les acteurs avec lesquels il est en relation d'héritage. Pour cela on définit la méthode ***RechercherRelationA(acteur)*** (Algorithme 5.13).

Cette méthode permet de créer l'ensemble des acteurs avec lesquels un acteur est en relation d'héritage.

Algorithme 5.13 La méthode *RechercherRelationA(acteur)*

RechercherRelationA(a : Acteur)

Début
  pour *chaque* $generalization_i \subset a$ **faire**
    si ($generalization_i.specialization$ *de type* $Actor$) **alors**
      $heritageEnsembleA = heritageEnsembleA \cup generalization_i.specialization$
    finSi
  fait
  retourner *heritageEnsembleA*
Fin

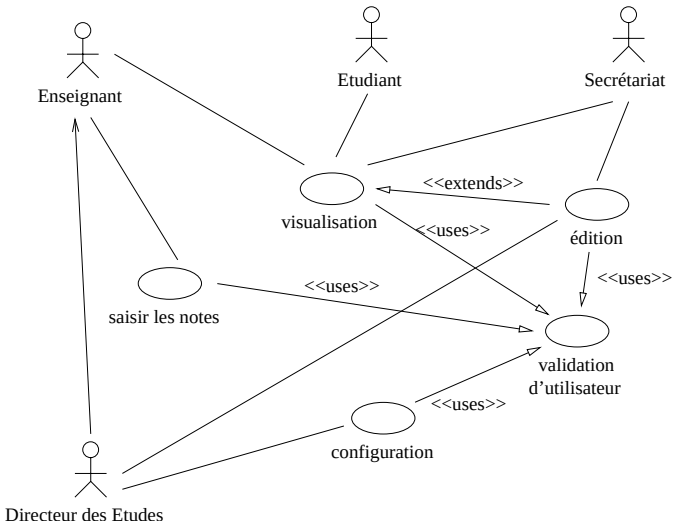


FIG. 5.3 – Un diagramme de cas d'utilisation d'une application "Gestion des Notes".

Le parcours du méta-modèle d'UML est similaire à celui utilisé dans la méthode *Rechercher-RelationCU(casUtilisation)*. Seulement cette fois la méthode prend comme argument un *acteur* et retourne l'ensemble des acteurs. De plus, un *Classifier* est une sous-classe de *GeneralizableElement*.

A partir de la classe *GeneralizableElement* en utilisant le rôle fils de l'association, on atteint l'ensemble des instances de la classe *Generalization*. On utilise pour chacune de ces instances le rôle *specialization* d'une association. Si cette instance est de type *Actor*, on l'ajout à l'ensemble *heritageEnsembleA*.

Nous avons spécifié l'algorithme de construction de rôles du système d'information à partir de la représentation des éléments du modèle RBAC en utilisant le méta-modèle d'UML. Cet algorithme a été implémenté en utilisant le langage de programmation Java.

5.2.2 Notre exemple d'application en UML

Nous avons choisi comme exemple une application "Gestion des Notes" qui peut être contenue dans le système d'information d'une université.

L'application a été écrite en utilisant les concepts du langage UML avec l'outil Rational Rose. On a créé le diagramme de cas d'utilisation principal qui contient tous les acteurs et les cas d'utilisation de l'application (Figure 5.3).

Chaque cas d'utilisation possède des collaborations pour lesquelles on a réalisé des diagrammes de séquence :

1. *Configuration* - les diagrammes de séquence : "Saisie d'un Enseignant", "Saisie d'un Etudiant", "Saisie d'une Matière", "Affectation d'un enseignant à une matière", "Affectation d'un étudiant à une matière",
2. *Validation d'utilisateur* - le diagramme de séquence : "Identification",
3. *Visualisation* - les diagrammes de séquence : "Visualiser du bulletin d'un étudiant", "Visualiser les notes obtenues pour un contrôle", "Visualiser toutes les notes des étudiants pour chaque matière", Visualiser les notes",
4. *Édition* - les diagrammes de séquence : "Édition du bulletin d'un étudiant", "Édition des notes obtenues pour un contrôle", "Édition de toutes les notes des étudiants pour chaque matière",
5. *Saisir les Notes* - le diagramme de séquence : "Saisie de Note".

Chaque diagramme de séquence contient la description du cas d'utilisation en utilisant des ensembles de messages et d'objets caractérisés pour chaque cas d'utilisation et chaque collaboration.

La description de cette application est présentée en détails dans l'Annexe A.

5.2.3 Le fichier XMI et le parseur XMI/XML

La deuxième étape est l'étape de production du fichier XMI. On a généré ce type de fichier en se basant sur l'application créée en UML. Le fichier XMI est crée automatiquement par l'AGL utilisé (Rational Rose ou Argouml).

Nous avons développé ensuite un parseur de fichiers XMI écrit en Java. Ce parseur utilise le méta-modèle d'UML pour analyser les fichiers de type XMI. Le fichier XMI obtenu contient la description dans la syntaxe XML de tous les éléments de chaque diagramme de notre application d'UML, i.e. "Gestion des Notes".

Nous utilisons ce parseur XMI/XML pour obtenir ensuite un arbre des éléments qui contient tous les éléments correspondant au modèle RBAC étendu.

On a utilisé deux types de diagrammes dans notre cas : des diagrammes de cas d'utilisation et des diagrammes de séquence. Les éléments de ces diagrammes sont structurés dans un fichier XMI basé sur le méta-modèle d'UML.

Les éléments de diagramme de cas d'utilisation (use case diagram) sont comme suit :

- *cas d'utilisation* (use case), par exemple *configuration*, *visualiser*.
- *acteur* (actor), par exemple *Enseignant*, *Étudiant*.
- *des relations* :
  - *relation de généralisation* - une relation entre les cas d'utilisation ou entre les acteurs ; pour les cas d'utilisation - les relations d'extension et les relation d'utilisation, pour les acteurs - les relations d'héritage.
  - *relation de communication* - une relation entre un acteur et un cas d'utilisation (relation d'association).

Les éléments de diagramme de séquence (sequence diagram) sont comme suit :

- *classe*, par exemple : *Liste d'Enseignants*.
- *objet*, un parseur XMI/XML donne :
  - un nom de cet objet,
  - un message qui est envoyé à cet objet
  - et un message qui est envoyé par cet objet et une classe de base de cet objet,
- *interaction* - une méthode réalisée par un objet ; le parseur XMI/XML donne pour cette interaction :
  - un nom d'un message qui est envoyé au cours de cette interaction,
  - un prédécesseur de cette interaction,
  - un activateur de cette interaction,
  - un objet qui a reçu ce message,
  - et un objet qui a envoyé ce message.

La fonctionnalité du parseur XMI/XML est accomplie par les étapes suivantes (Figure 5.4) :

- le parcours du fichier XMI et l'identification des éléments qui existent dans les deux types de diagrammes (i.e. diagramme de cas d'utilisation et diagramme de séquence) : acteurs, cas d'utilisation, leurs relations, classes, objets et messages envoyés entre ces objets,
- la définition des ensembles des éléments identifiés,
- l'identification de ces éléments comme les éléments caractéristiques pour les concepts du modèle RBAC étendu,
- la détermination des ensembles des éléments de l'application comme les ensembles des éléments caractéristiques pour le modèle RBAC étendu,

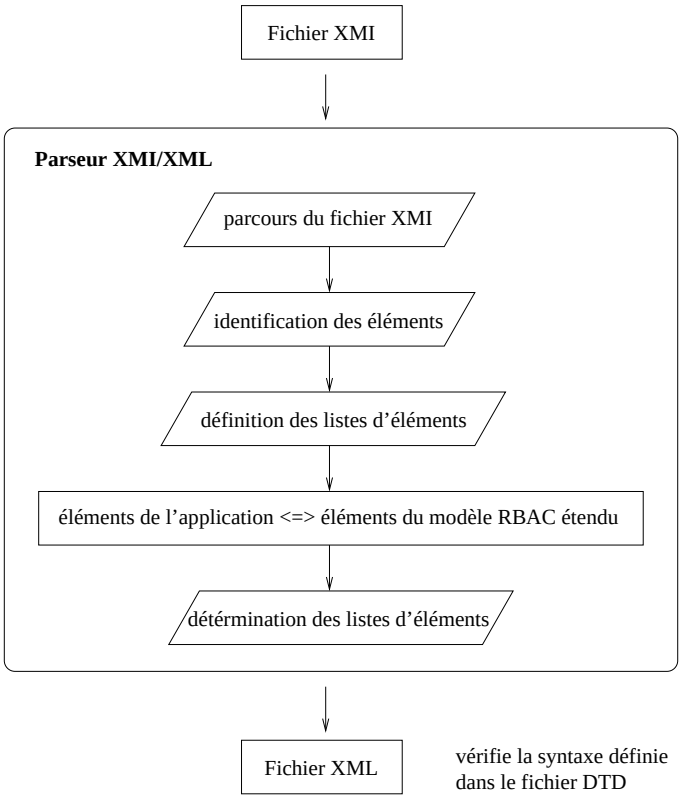


FIG. 5.4 – Les étapes de la fonctionnalité du parseur XMI/XML.

– la génération du fichier XML qui contient les ensembles des éléments déterminés.

Un utilisateur de ce parseur obtient un fichier au format XML. Ce fichier contient tous les éléments et leurs propriétés de l'application d'UML qui sont ensuite utilisés dans une intégration du modèle RBAC. Il contient la description des éléments du modèle RBAC étendu : des rôles, des fonctions, des permissions, des objets et des méthodes.

5.2.4 Le document résultat

Un fichier DTD est un document DTD qui permet de spécifier le syntaxe d'un document XML valide dans un fichier particulier. Le fichier DTD est nommé *RBAC.dtd* et il décrit la structure du modèle RBAC étendu présenté dans le Chapitre 3. Il contient tous les éléments qui sont caractéristiques pour ce modèle en utilisant la notion du langage XML.

Le parseur génère un fichier XML. Il est nommé *RBAC_NomApplication.xml*. Ce fichier est créé automatiquement après l'exécution du parseur XMI/XML qui prend comme entrée un fichier XMI décrivant une application UML, e.g. "Gestion des Notes" - *RBAC_GestionDesNotes.xml*.

La description formelle des éléments d'un fichier RBAC_NomApplication.xml existe dans le fichier RBAC.dtd. L'élément principal dans le fichier RBAC.dtd s'appelle ***RBAC***. Il contient un ou plusieurs éléments fils qui sont : des rôles, des fonctions, des permissions, des méthodes, des objets, des opérations et des classes. Il est décrit comme suit [PGH01] :

<!ELEMENT RBAC(role+, fonction, permission*, methode*, objet*, operation*, classe*)>*

Chaque élément fils de l'élément *RBAC* est ensuite décrit en accord avec le modèle RBAC étendu. La structure du fichier RBAC.dtd et la description des éléments de cette structure sont

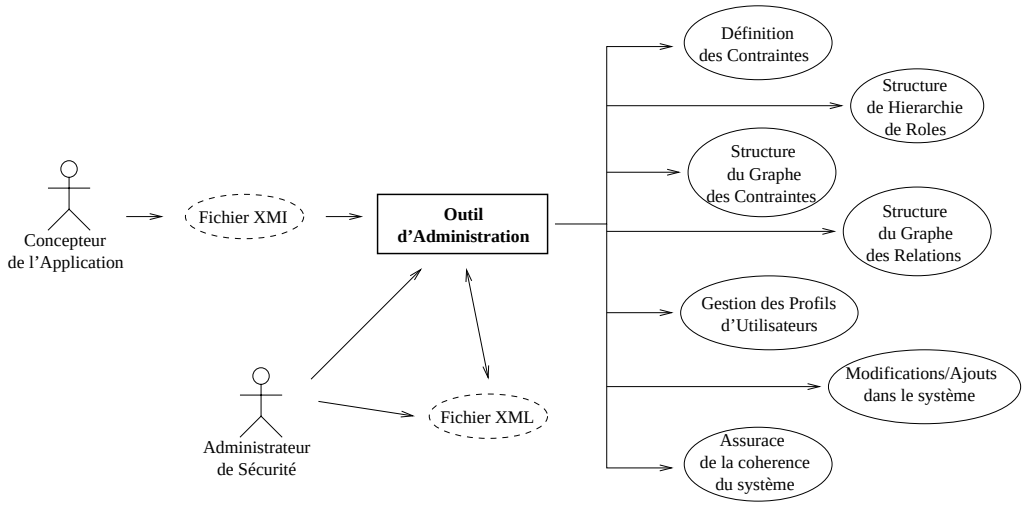


FIG. 5.5 – L'outil de l'administrateur de sécurité.

présentées dans l'Annexe E.

Le fichier RBAC_GestionDesNotes.xml est produit à partir de l'application "Gestion des Notes". Il contient les éléments concrets qui existent dans cette application du système d'information et qui représentent les utilisateurs de ce système, les rôles, les fonctions, etc.

La description du fichier RBAC_GestionDesNotes.xml qui a été généré pour l'application "Gestion des Notes" est présentée dans l'Annexe E.

5.3 L'outil de l'administrateur de sécurité

On propose dans ce mémoire un outil qui permettra à l'administrateur de sécurité de gérer la sécurité du système d'information au niveau du contrôle d'accès. La Figure 5.5 présente les fonctions de l'outil d'administration de la plate-forme UML-XML-Java.

5.3.1 La réalisation des tâches de l'administrateur de sécurité

Le document résultat (décrit dans la section 5.2.4) est le produit du travail d'un concepteur au niveau contrôle d'accès du système d'information. Il contient les éléments de l'application créée par le concepteur traduits en concepts du modèle RBAC étendu.

Ce document est donné ensuite à l'administrateur de sécurité du système d'information qui gère la sécurité de tout le système. Il doit intégrer les nouveaux éléments, e.g. la nouvelle application, dans le système au niveau sécurité. Le document XML permet de réaliser cette intégration indépendamment de l'environnement. Il décrit les éléments de la nouvelle application au niveau du modèle RBAC utilisé pour l'administration du contrôle d'accès dans le système.

L'administrateur de sécurité, qui réalise l'étape d'exploitation du système (présenté dans le Chapitre 4, la section 2.5.2) se sert du document XML pour définir les privilèges des utilisateurs dans le système. Il détermine ainsi les droits des utilisateurs pour utiliser chaque application du système décrite dans son document XML.

L'administrateur de sécurité doit réaliser les tâches suivantes pour chaque nouveau élément, e.g. nouvelle application, ajouté dans le système d'information :

– définir les contraintes au niveau de l'administration (présenté dans le Chapitre 4, les sections 2.3 et 2.4),

- ajouter les nouveaux rôles de la nouvelle application dans une hiérarchie de rôles du système,
- ajouter les contraintes de la nouvelle application dans un graphe des contraintes du système,
- ajouter les relations entre les éléments de la nouvelle application et les relations entre les éléments du système et de la nouvelle application dans le graphe des relations en accord avec les contraintes définies,
- définir les profils utilisateur, i.e. définition des ensembles de rôles pour chaque utilisateur qui va utiliser la nouvelle application.

Ces actions doivent être vérifiées pour garantir la cohérence globale de la sécurité. La plate-forme doit réaliser la validation de l'intégration des travaux du concepteur de l'application et de l'administrateur de sécurité. La validation garantit l'intégration finale de la nouvelle application au niveau du contrôle d'accès du système d'information.

5.3.2 Les perspectives de l'outil de l'administrateur

Les actions de l'outil de l'administrateur, présentées dans la section précédente, ne sont pas toutes implémentées pour le moment.

L'outil présenté implémente seulement le modèle RBAC jusqu'au modèle $RBAC_2$, c'est-à-dire il n'implémente pas encore de modèle $RBAC_3$ (le niveau des contraintes). La représentation de ce niveau fait aussi partie du travail futur sur la plate-forme.

Donc, les travaux à faire pour terminer la plate-forme sont les suivants :

- l'implémentation de toutes les actions de l'outil d'administration et les modifications éventuelles,
- la réalisation du niveau contrainte du modèle RBAC étendu sur la plate-forme - tous les concepts décrits dans le Chapitre 4,
- l'implémentation de l'algorithme de vérification de la cohérence du système définie dans le chapitre précédent.

5.4 Conclusion

L'objectif de ce chapitre était de présenter notre plate-forme pour la création des rôles du système d'information au niveau du contrôle d'accès.

Nous avons introduit, dans un premier temps, les outils utilisés pour la réalisation de cette plate-forme. Ensuite, nous avons présenté l'algorithme de la création des rôles que nous avons implémenté. Cet algorithme se compose de deux étapes et se base sur le rapprochement des concepts UML et RBAC étendu présenté dans le Chapitre 3.

L'outil d'administration, présenté dans la dernière partie du chapitre, est actuellement en cours de développement. Il permettra à l'administrateur de sécurité de gérer le contrôle d'accès d'un système information par la création du profil des utilisateurs de ce système. Cet outil implémentera les concepts présentés dans les deux chapitres précédents.

Conclusions et perspectives

De nos jours avec la mondialisation des marchés et l'accélération des échanges d'information qui en découlent, le système d'information d'une entreprise devient de plus en plus stratégique pour la pérennité de celle-ci. Il importe donc de sécuriser efficacement celui-ci. Une facette de la sécurité consiste à assurer l'accès logique aux différents composants (i.e. données, services) du système d'information.

Le modèle RBAC permet de mettre en place une politique de contrôle d'accès plus de flexibilité dans l'organisation d'un schéma de contrôle d'accès associé au système d'information, c'est pourquoi nous l'avons choisi. Malheureusement, malgré de nombreux modèles RBAC proposés, peu de travaux se sont intéressés à la mise en place effective d'un modèle RBAC, c'est à dire à l'ingénierie de conception des rôles. De plus nous avons remarqué que la conception d'un système d'information est peu interconnectée avec celle de son schéma de contrôle d'accès associé.

C'est précisément le domaine que nous avons étudié dans cette thèse.

Nous avons choisi d'aborder le problème du contrôle d'accès d'un système d'information en proposant un modèle de rôles dès la conception de celui-ci et tout au long de son évolution (i.e. ajouts de nouvelles applications). Nos objectifs étaient d'une part de faciliter le travail de l'administrateur de sécurité et d'autre part d'avoir une meilleure cohérence entre les contraintes globales de sécurité de l'entreprise et les différents composants de son système d'information.

Pour ce faire, nous avons utilisé une conception orientée objet décrite dans le langage UML. Grâce à ces diagrammes de haut niveau, UML permet de spécifier clairement les besoins du système d'information et facilite le dialogue entre les différents acteurs (i.e. concepteur, utilisateur, administrateur ...). C'est pourquoi nous l'avons privilégié et de plus il est devenu depuis le démarrage de cette thèse un standard de facto dans la communauté objet.

Nous avons pris le parti de décomposer le processus de création des rôles en deux étapes correspondant à un partage des responsabilités au sein d'une coopération entre le concepteur d'un composant du système d'information qui définit les rôles, car c'est le plus apte à connaître les contraintes de l'application, et l'administrateur de sécurité qui les affecte aux utilisateurs finals, car c'est le plus apte à vérifier les contraintes globales de l'entreprise. Une étape supplémentaire de concertation entre les deux parties (concepteur et administrateur) est nécessaire dans le cas l'apparition de conflit ou d'incohérence dans le système global lié à une modification de celui-ci.

A partir donc d'une spécification UML fournie par un concepteur, nous avons montré comment il était possible de générer automatiquement les rôles associés à un composant du système d'information ceci en utilisant le méta-modèle d'UML. Pour cela nous avons rapproché certains concepts d'UML avec ceux de notre modèle RBAC étendu. La tâche de l'administrateur consiste dès lors à affecter les différents utilisateurs aux rôles en fonction de leur position dans l'entreprise et des contraintes globales de sécurité.

Puis pour affiner notre modèle RBAC, nous avons ensuite intégré la notion de contrainte d'accès dans les diagrammes UML à l'aide du langage OCL pour spécifier des contraintes applicatives et nous avons utilisé au niveau de l'administrateur de sécurité le langage RCL 2000 pour spécifier des contraintes plus globales. Nous avons proposé un algorithme pour unifier l'ensemble des contraintes

vers le langage RCL 2000 et un autre algorithme pour vérifier une cohérence "minimale" du modèle RBAC lors de l'intégration d'un nouveau composant dans un système d'information existant.

Une plate-forme UML-XML-Java utilisant l'outil de Rational Rose a été développée, dans une première version sans les contraintes, pour générer un modèle RBAC étendu.

Pour résumer, les apports nouveaux de la thèse sont les suivants :

- l'extension d'un modèle du contrôle d'accès RBAC avec l'introduction d'une plus grande flexibilité dans le partage et la répartition des responsabilités des utilisateurs du système d'information,
- l'utilisation du langage UML pour l'implémentation d'un modèle du contrôle d'accès basé sur la notion de rôle et ceci pendant la conception du système d'information,
- l'élaboration d'une méthode automatique de création des rôles d'un système d'information et de leurs permissions associées, par un rapprochement des concepts du modèle RBAC étendu avec ceux définis dans le langage UML,
- un partage des responsabilités au sein d'une coopération entre le concepteur d'un composant du système d'information et l'administrateur de sécurité dans le processus de définition des rôles et des contraintes d'accès associées (i.e. applicatives ou globales),
- la définition de contraintes d'accès applicatives en utilisant le langage OCL et sa traduction vers le langage RCL 2000, créé spécialement pour représenter les contraintes du modèle RBAC,
- la vérification d'une cohérence globale (minimale ou faible) du modèle de contrôle d'accès lors de l'intégration d'un nouveau composant dans le système d'information,
- la génération automatique d'une liste de rôles dans un formalisme XML, indépendant de la plate-forme finale.

Les perspectives de nos travaux de recherche sont les suivantes :

1. D'un point de vue théorique, il serait intéressant d'introduire une notion de cohérence forte en exprimant chaque nouveau composant à l'aide d'un ensemble de contraintes qui traduirait les nouveaux éléments (i.e. entités, relations et contraintes) à introduire dans le modèle RBAC existant. Ainsi à l'aide d'un outil de type CSP pour valider la cohérence globale nous pourrions détecter tout élément nouveau posant problème avec le modèle RBAC existant.
2. D'un point de vue applicatif, il nous faut d'une part compléter la plate-forme UML-XML-Java de génération de rôles pour y intégrer la génération des contraintes d'accès associées. D'autre part, il nous faut également commencer l'écriture de l'outil d'administration des rôles avec une phase de vérification de la cohérence "faible" ou "forte".

Annexes

Annexe A

L'application "Gestion des Notes"

A.1 Analyse du problème

L'application "Gestion des Notes" est un élément du système d'information décrivant le système de l'université. Cette application permet de la gestion des notes des étudiants de chaque formation de l'université.

Ce système de gestion de notes doit permettre de saisir les notes des étudiants, de calculer les les moyennes et d'imprimer les bulletins de notes.

A.1.1 Le cahier de charges

Chaque formation consiste en plusieurs années d'études composées de deux semestres. Chaque semestre, les contrôles de connaissances sont établis pour chaque matière, les enseignants vont les évaluer et le système doit pouvoir répondre aux besoins d'édition de la liste des notes et des bulletins de notes.

Cette application permet des actions suivantes :

- Les enseignants peuvent visualiser la liste des étudiants pour la matière qu'ils enseignent et les évaluer selon les contrôles réalisés ; seul l'enseignant responsable de la matière peut saisir les notes. Plusieurs enseignants peuvent enseigner la même matière. Ils ne peuvent uniquement modifier que les notes qu'ils ont données, mais ils peuvent visualiser toutes les notes des étudiants quelque soit la matière.
- Les étudiants peuvent visualiser leur bulletin de notes. Ils ne peuvent visualiser uniquement que leurs notes et visualiser leur moyennes obtenues sur les matières pour lesquelles ils ont été contrôlés.
- Le secrétariat peut imprimer et visualiser les bulletins de notes, les listes des notes et la liste des notes des étudiants pour un contrôle,
- Le directeur des études peut configurer le système, i.e. saisir un enseignant, saisir un étudiant, saisir une matière, affecter un enseignant à une matière et affecter un étudiant à une matière. D'autre part, il a le droit de modifier les notes des étudiants et de visualiser et imprimer toutes les listes que le Secrétariat peut visualiser et imprimer.

Tous les utilisateurs peuvent accéder à cette application du système d'information après identification et authentification par le système - le cas d'utilisation *Validation d'utilisateur*.

A.1.2 Les cas d'utilisation principaux de l'application

Les ***acteurs*** de l'application :

- *Etudiant* - une personne qui est évaluée par le contrôle,
- *Enseignant* - une personne qui évalue les étudiants selon les contrôles.
- *Secrétariat* - une personne qui imprime les bulletins de notes et la liste des notes.

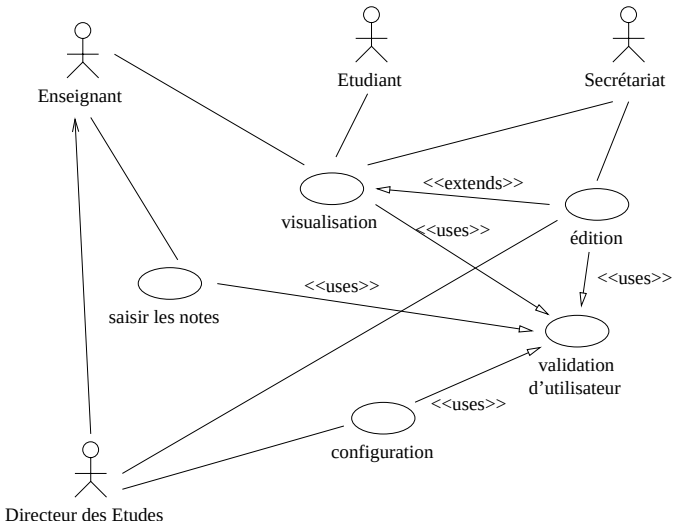


FIG. A.1 – Le diagramme de cas d'utilisation "Gestion des Notes".

– *Directeur des Etudes* - une personne qui administre les éléments de l'application.

Les ***cas d'utilisation*** de l'application :
– L'acteur Enseignant interagit avec le système pour visualiser des listes de notes (modifier des notes), visualiser des contrôles, créer des contrôles et les modifier.
– L'acteur Etudiant utilise le système pour visualiser ses propres notes et son propre bulletin de notes.
– L'acteur Secrétariat utilise le système pour la visualisation et l'édition des bulletins des étudiants et la liste des notes pour chaque matière.
– L'acteur Directeur des Etudes est responsable de la configuration de l'application du système, il est responsable de la saisie de notes. Il peut visualiser et éditer les listes des notes et les bulletins.

A.2 Les diagrammes d'UML de l'application

A.2.1 Les diagrammes de cas d'utilisation

A.2.1.1 Le diagramme principale

La Figure A.1 présente le diagramme de cas d'utilisation qui décrit les acteurs et les cas d'utilisation principaux de l'application "Gestion des Notes".

A.2.1.2 Le diagramme de cas d'utilisation "visualisation"

Le cas d'utilisation *visualiser* est une généralisation de différents types de visualisation (Figure A.2). L'acteur *Etudiant* peut exécuter les cas d'utilisation *visualiser le bulletin* et *visualiser les notes*. Les autres acteurs peuvent exécuter tous les cas d'utilisation de ce diagramme. La relation d'extension est utilisée pour cette généralisation au sens de la spécialisation.

A.2.1.3 Le diagramme de cas d'utilisation "édition"

Le cas d'utilisation *édition* est une extension de *visualisation* et il est une généralisation de différents types d'édition (Figure A.3). Les acteurs *Directeur des Etudes* et *Secrétariat* peuvent imprimer les listes et les bulletins. La relation d'extension est utilisée pour spécialiser édition.

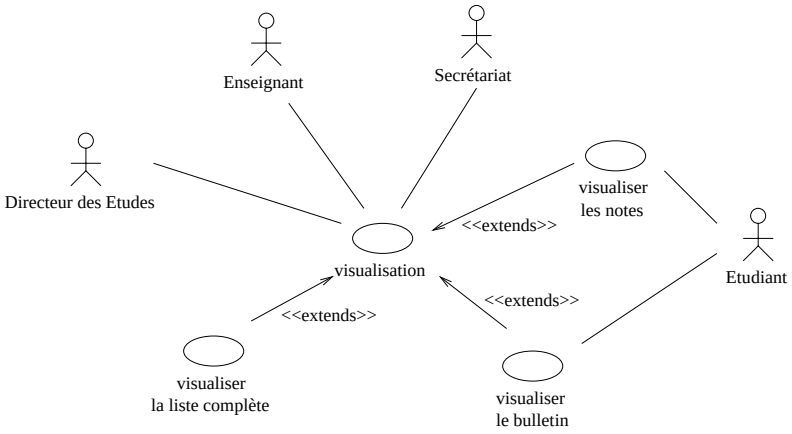


FIG. A.2 – Le diagramme de cas d'utilisation "visualisation".

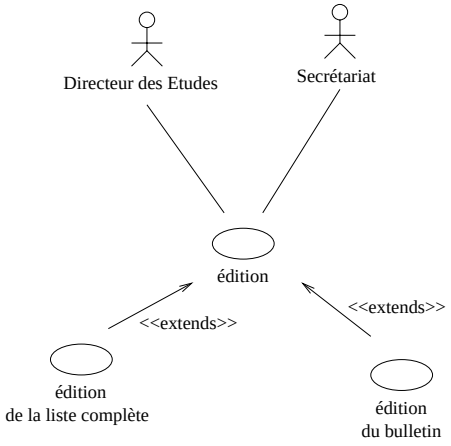


FIG. A.3 – Le diagramme de cas d'utilisation "édition".

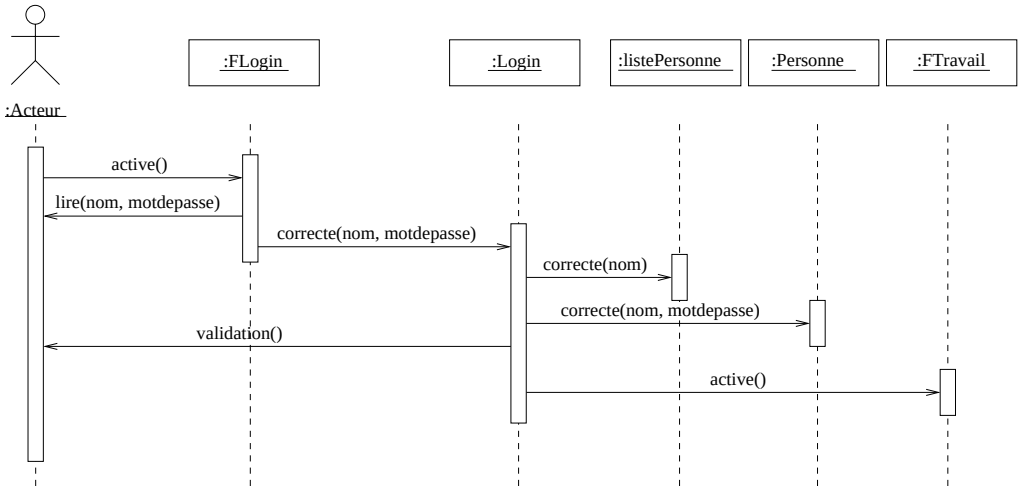


FIG. A.4 – Le diagramme de séquence "Identification".

A.2.2 Les scénarios

Les principaux scénarios de l'application sont les suivants :
- *Configuration* - réalisé par le *Directeur des Etudes*, consiste en :
  - "Saisie d'un Enseignant",
  - "Saisie d'un Etudiant",
  - "Saisie d'une Matière",
  - "Affectation d'un enseignant à une matière",
  - "Affectation d'un étudiant à une matière",
- *Validation d'utilisateur* - réalisé par chaque acteur de l'application, consiste en "Identification",
- *Visualisation* - réalisé par chaque acteur, consiste en :
  - "Visualiser le bulletin d'un étudiant",
  - "Visualiser les notes obtenues pour un contrôle",
  - "Visualiser toutes les notes des étudiants pour chaque matière",
  - "Visualiser les notes",
- *Edition* - réalisé par le Directeur des Etudes et le Secrétariat, consiste en :
  - "Edition du bulletin d'un étudiant",
  - "Edition des notes obtenues pour un contrôle",
  - "Edition de toutes les notes des étudiants pour chaque matière",
- *Saisir les Notes* - exécuté par l'Enseignant et par le Directeur des Etudes, consiste en : "Saisie de Note".

Les scénarios présentés ci-dessous sont décrits par les diagrammes de séquences correspondants.

A.2.3 Les diagrammes de séquence

A.2.3.1 Validation d'utilisateur

Le diagramme de séquence *Identification* est réalisé par chaque acteur, i.e. *Enseignant*, *Etudiant*, *Directeur des Etudes* et *Secrétariat*. Chacun est représenté par un Acteur dans le diagramme (Figure A.4).

Le cas d'utilisation *Validation d'utilisateur* est réalisé par chaque utilisateur qui veut accéder à l'application. Chaque objet du domaine de l'interface d'utilisateur de l'application est visualisé par un objet d'une classe de l'interface, préfixé par la lettre "F" qui signifie une "fenêtre".

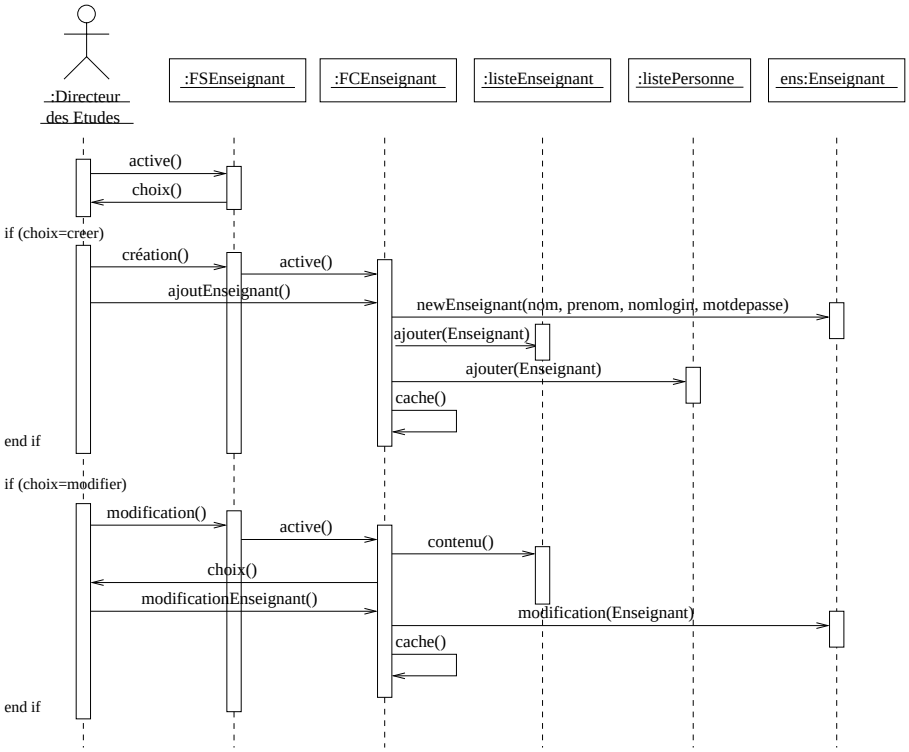


FIG. A.5 – Le diagramme de séquence "Saisie d'un Enseignant".

Dans ce diagramme *FLogin* est l'objet graphique qui permet d'accéder à l'objet de la classe *Login*. La fenêtre de travail (*FTravail*) est la fenêtre globale qui apparaît pour chaque utilisateur du système, qui a été identifié et authentifié.

A.2.3.2 Configuration

Tous les diagrammes de séquence de ce cas d'utilisation sont exécutés par le *Directeur des Etudes*. Ils sont initialisés dans la fenêtre *FConfiguration* qui représente un objet graphique pour accéder aux objets de la classe *Configuration*.

Saisie d'un Enseignant

La Figure A.5 présente le diagramme de séquence pour créer un nouvel *Enseignant* ou modifier certaines informations de l'enseignant déjà créé. Le Directeur des Etudes possède donc deux possibilités ("*choix=créer*" ou "*choix=modifier*").

On utilise deux fenêtres d'affichage : *FSEnseignant* - pour saisir d'un enseignant et *FCEnseignant* - pour définir les informations d'un enseignant. Chaque ajout d'un enseignant provoque l'ajout dans la liste des enseignants et dans la liste des personnes pour garantir son identification et son authentification.

Saisie d'un Etudiant

La Figure A.6 présente le diagramme de séquence pour créer un nouvel *Etudiant* ou modifier certaines information d'un étudiant déjà créé. On utilise deux fenêtres d'affichage : *FSEtudiant* - pour saisir d'un étudiant et *FCEtudiant* - pour définir les informations d'un étudiant. Chaque ajout d'un étudiant provoque l'ajout dans la liste des étudiants et dans la liste des personnes pour garantir son identification et son authentification.

Saisie d'une Matière

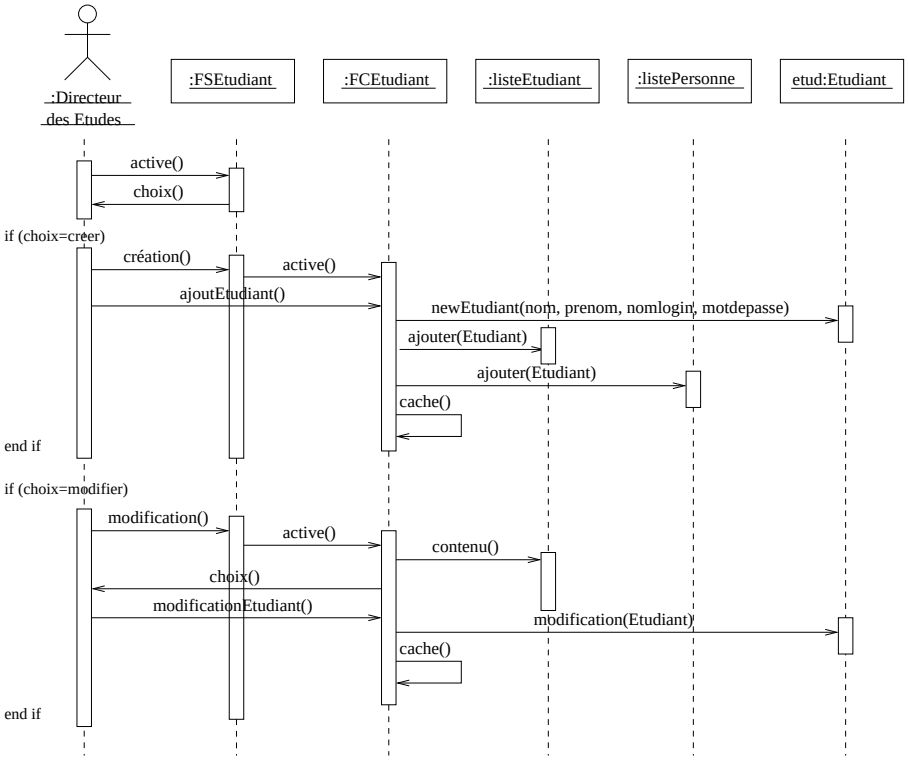


FIG. A.6 – Le diagramme de séquence "Saisie d'un Etudiant".

La Figure A.7 présente le diagramme de séquence pour créer une nouvelle *Matière* ou modifier certaines information d'une matière déjà créée. On utilise deux fenêtres d'affichage : *FSMatière* - pour saisir d'une matière et *FCMatière* - pour définir les informations d'une matière.

Affectation d'un Enseignant à une Matière

La Figure A.8 présente le diagramme de séquence pour affecter un *Enseignant* à une *Matière*. Le *Directeur des Etudes* choisit un enseignant de la liste des enseignants et choisit une matière de la liste des matières. L'application ajoute l'enseignant à la liste des enseignants de la matière en supposant qu'une matière peut être enseignée par plusieurs enseignants. On attache la matière à la liste des matières de l'enseignant en supposant qu'un enseignant peut enseigner quelques matières.

Affectation d'un Etudiant à une Matière

La Figure A.9 présente le diagramme de séquence pour affecter un *Etudiant* à une *Matière*. Le *Directeur des Etudes* choisit un étudiant de la liste des étudiants et choisit une matière de la liste des matières. L'application ajoute l'étudiant à la liste des étudiants de la matière et la matière à la liste des matières de l'étudiant.

A.2.3.3 Visualisation

Chacun des acteurs qui participent dans les diagrammes de séquence, i.e. *Enseignant*, *Etudiant*, *Directeur des Etudes* et *Secrétariat*, est représenté par un Acteur.

Visualiser le bulletin d'un étudiant

Ce cas d'utilisation de visualiser le bulletin des notes d'un étudiant peut être réalisé par chaque acteur de l'application (Figure A.10).

La seule restriction intervient pour l'étudiant, car l'étudiant ne peut récupérer que son propre bulletin de notes. La fenêtre *FAfficherNotesEtudiant* est la fenêtre qui visualise les bulletins de

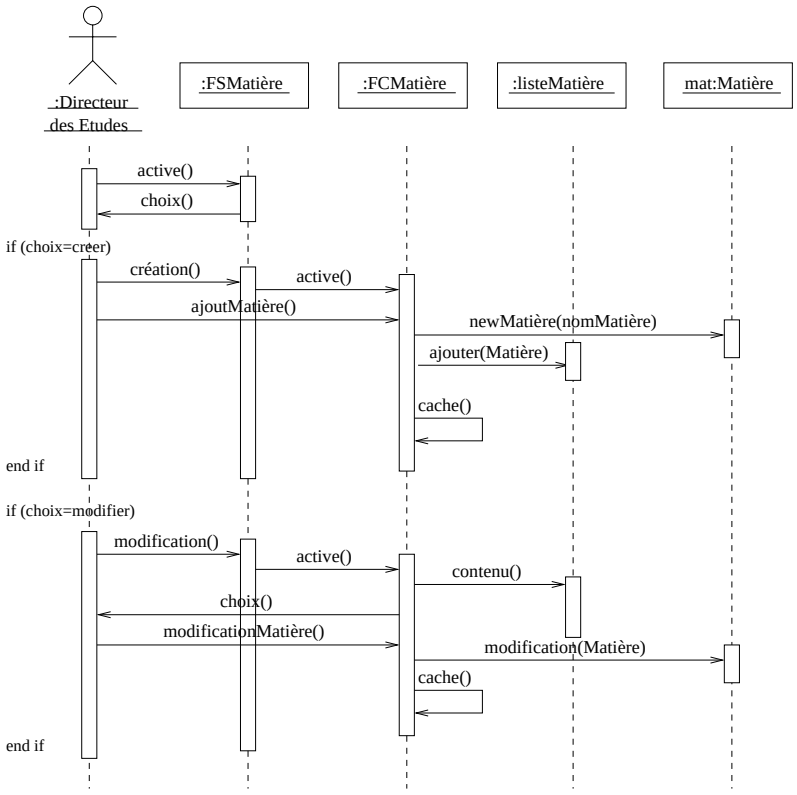


FIG. A.7 – Le diagramme de séquence "Saisie d'une Matière".

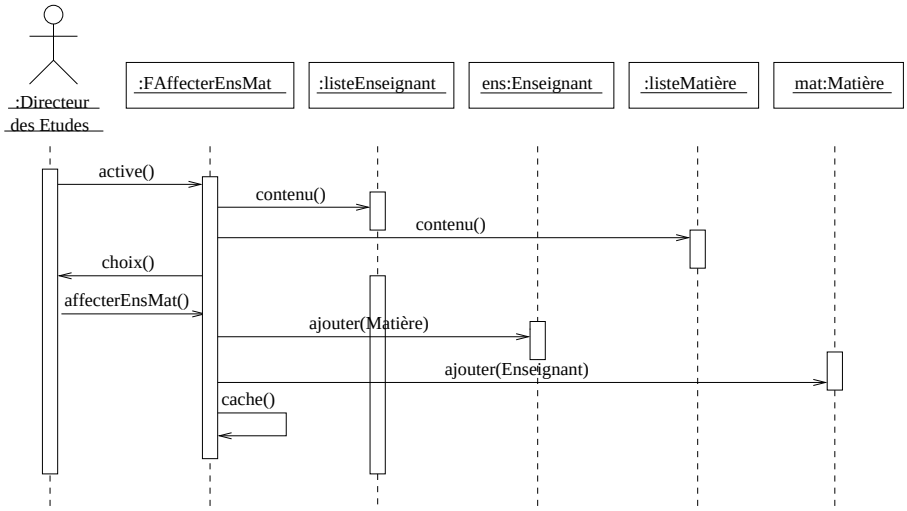


FIG. A.8 – Le diagramme de séquence "Affectation d'un Enseignant à une Matière".

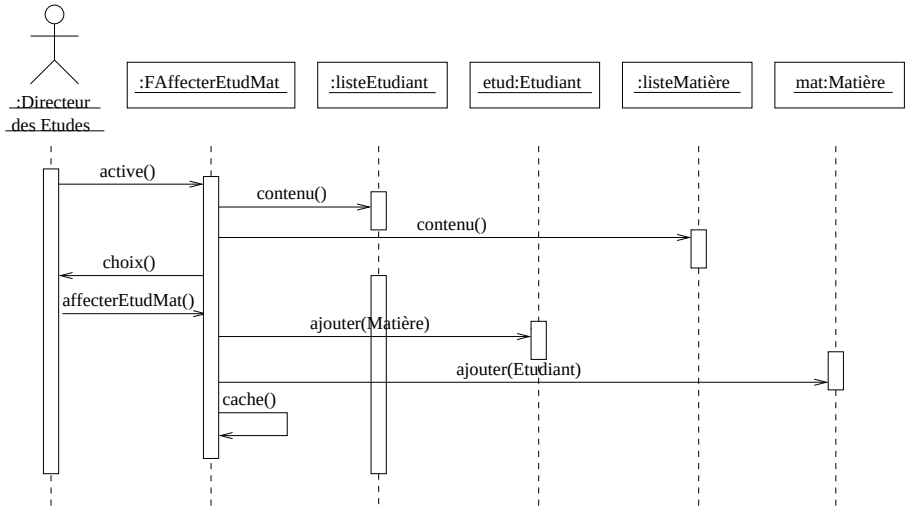


FIG. A.9 – Le diagramme de séquence "Affectation d'un Etudiant à une Matière".

notes des étudiants pour tous les utilisateurs de l'application. La fenêtre *FAfficherListeEtudiant* affiche la liste des étudiants. Il y a une contrainte pour la méthode *getValeur()* pour indiquer que l'étudiant ne peut visualiser que son propre bulletin des notes.

Visualiser les notes obtenues pour un contrôle

Ce cas d'utilisation est réalisé par tous les acteurs sauf par l'Etudiant (Figure A.11). La fenêtre *FAfficherControleEtudNote* permet de visualiser les notes avec les étudiants pour le contrôle choisi. La fenêtre *FAfficherListeControle* permet d'afficher la liste de tous les contrôles. Un utilisateur choisit la matière, puis le contrôle qu'il veut visualiser. Il obtient la liste des étudiants et la liste de notes pour le contrôle choisi - il obtient la note qu'un étudiant a reçu pour le contrôle. Cette opération est réalisée pour chaque étudiant qui a passé ce contrôle.

Visualiser toutes les notes des étudiants pour chaque matière

La visualisation de toutes les notes des étudiants pour chaque matière est un scénario ressemblant à la visualisation d'un bulletin de notes de façon récursive pour chaque étudiant (*while(étudiant)* et *while(matière)*).

Visualiser les notes

La Figure A.12 présente le diagramme de séquence qui permet de visualiser les notes obtenues par un étudiant dans un contrôle d'une matière. Pour obtenir la note, un utilisateur doit choisir : un étudiant, une matière et un contrôle.

A.2.3.4 Edition

Chacun des acteurs qui participent dans les diagrammes de séquence, i.e. *Enseignant*, *Directeur des Etudes* et *Secrétariat*, est représenté par un Acteur. L'acteur *Etudiant* ne participe pas dans ce cas d'utilisation.

Edition du bulletin d'un étudiant

Ce cas d'utilisation d'édition d'un bulletin de notes d'un étudiant peut être réalisé par chaque acteur de l'application sauf par l'Etudiant (Figure A.13).

Dans ce cas d'utilisation, la fenêtre *FAfficherNotesEtudiant* est la fenêtre qui visualise les bulletins de notes des étudiants. La fenêtre *FAfficherListeEtudiant* affiche la liste des étudiants.

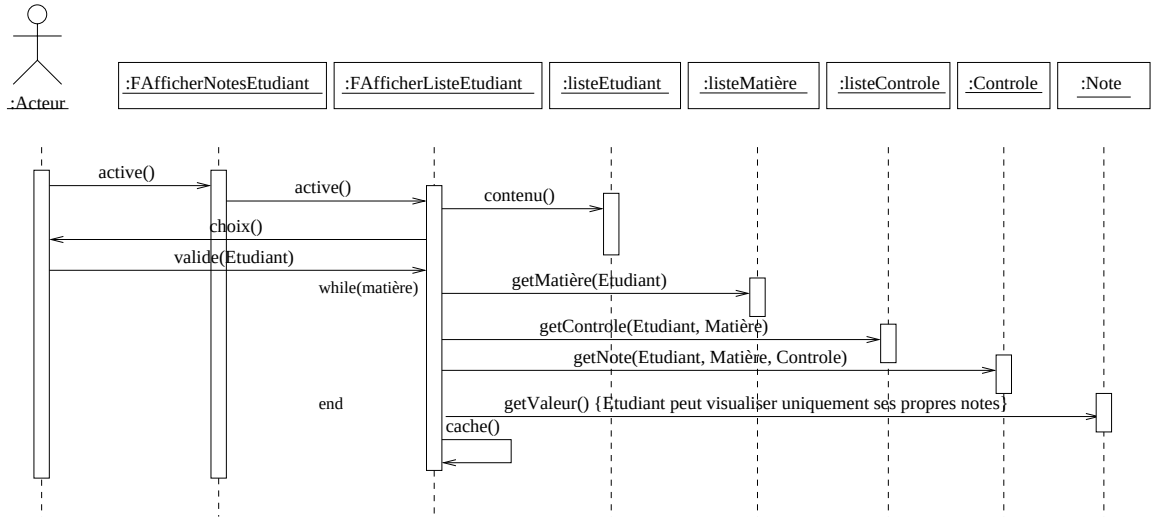


FIG. A.10 – Le diagramme de séquence "Visualiser le bulletin d'un étudiant".

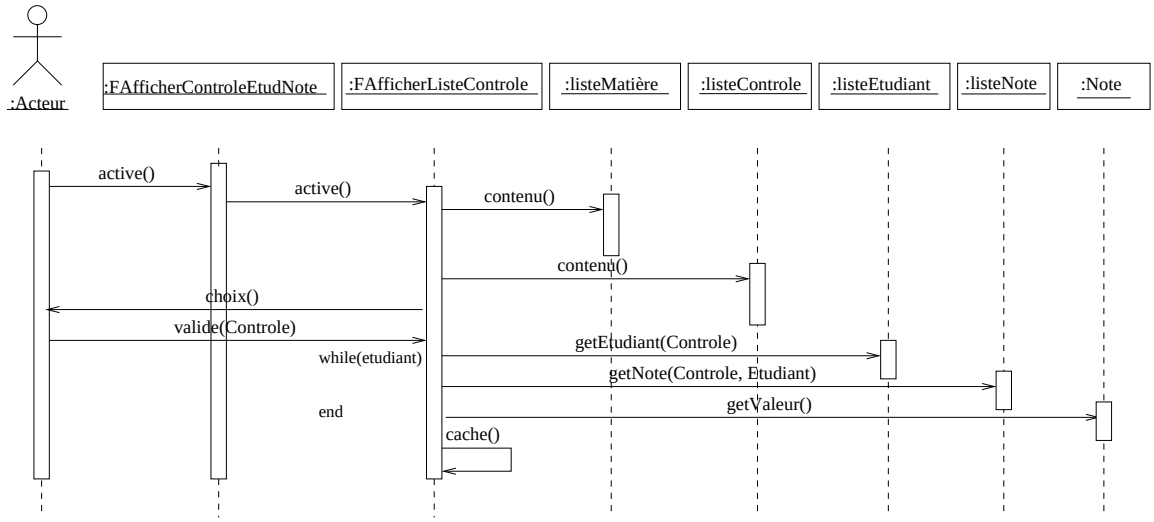


FIG. A.11 – Le diagramme de séquence "Visualiser les notes obtenues pour un contrôle".

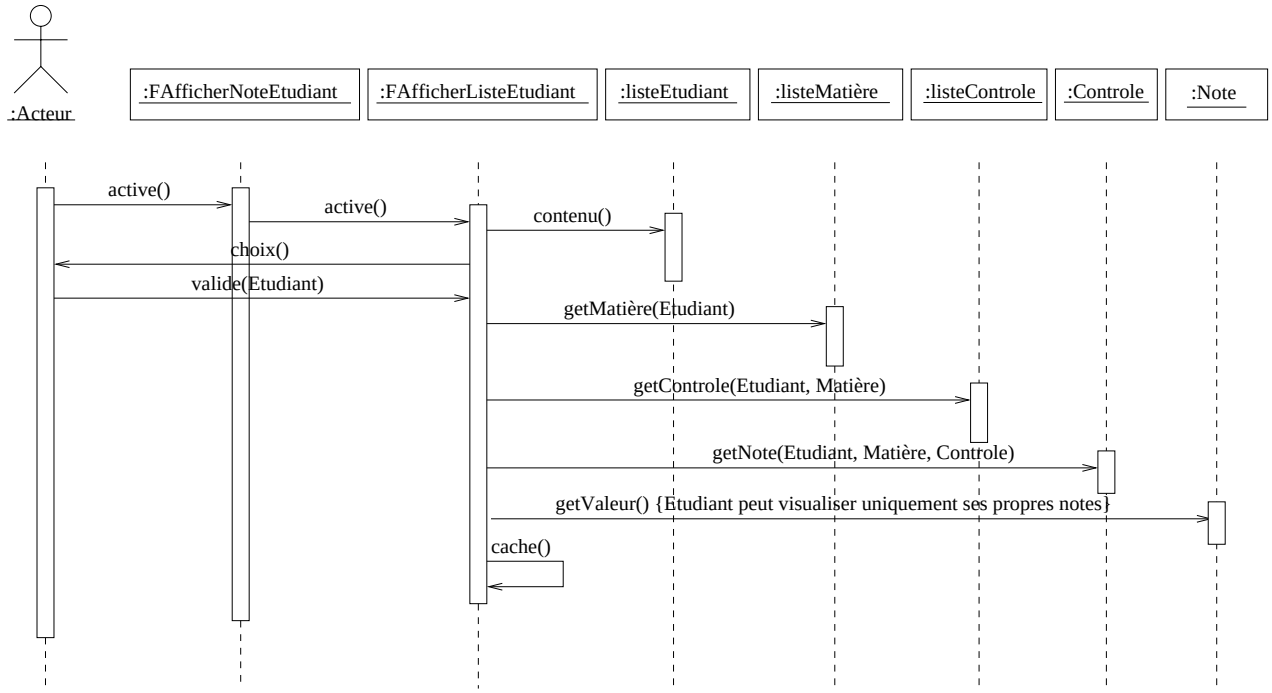


FIG. A.12 – Le diagramme de séquence "Visualiser les notes".

Puis à partir d'un choix "*imprimer*" l'application permet de l'édition du bulletin des notes de l'étudiant.

Edition des notes obtenues pour un contrôle

Ce cas d'utilisation est réalisé par tous les acteurs sauf par l'Etudiant (Figure A.14). Un utilisateur choisit la matière, puis le contrôle qu'il veut éditer. Il obtient la liste des étudiants et la liste des notes pour le contrôle choisi - il obtient la note qu'un étudiant a reçu pour le contrôle. Cette opération est réalisée pour chaque étudiant du contrôle.

Edition de toutes les notes des étudiants pour chaque matière

L'édition de toutes les notes des étudiants pour chaque matière est un scénario ressemblant à l'édition d'un bulletin de notes de façon récursive pour chaque étudiant (*while(étudiant)* et *while(matière)*).

A.2.3.5 Saisir les Notes

Le diagramme de séquence *Saisie de Note* est réalisé par *Enseignant* ou par *Directeur des Etudes*. Chacun est représenté par un Acteur dans le diagramme (Figure A.15).

Pourtant les enseignants ont une contrainte pour l'utilisation de ce cas d'utilisation : un enseignant ne peut créer ou modifier que les contrôles qu'il a effectués.

A.2.4 Le diagramme de classes

La Figure A.16 présente le diagramme de classes de l'application "Gestion des Notes".

Ce diagramme contient seulement les classes de type persistant, c'est-à-dire les objet peuvent être stockés dans une base de données du système.

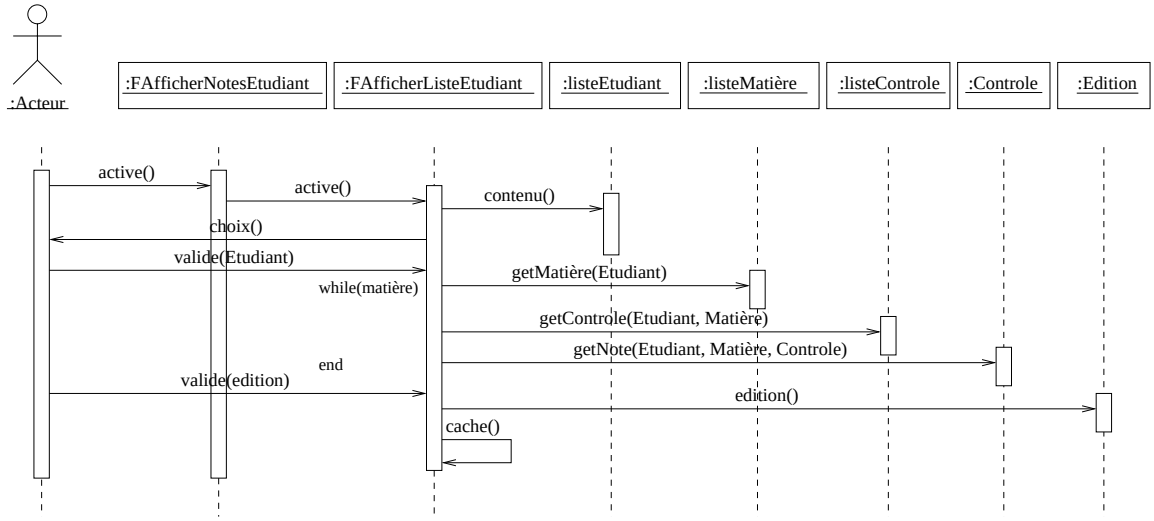


FIG. A.13 – Le diagramme de séquence "Edition du bulletin d'un étudiant".

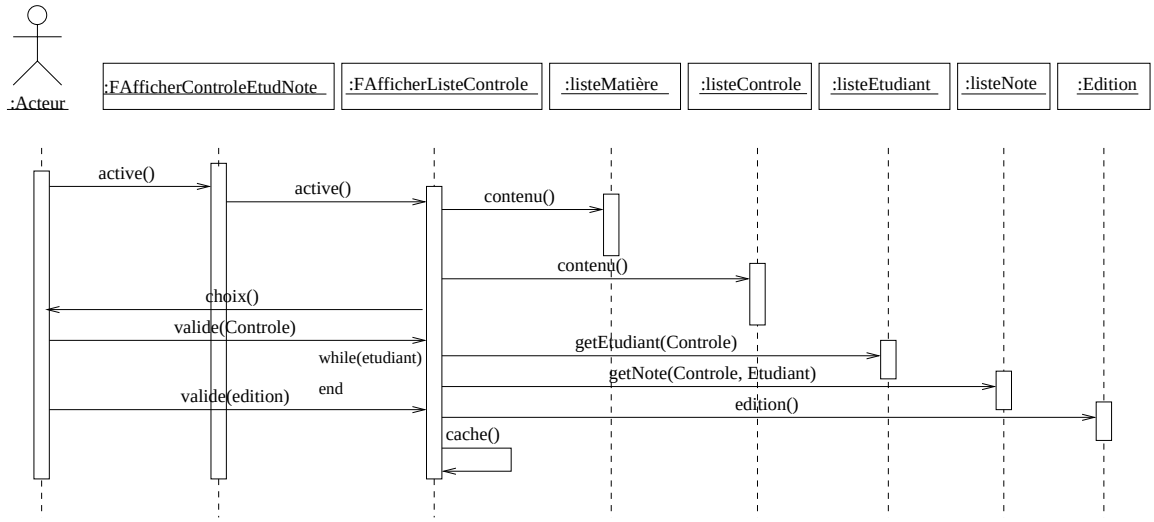


FIG. A.14 – Le diagramme de séquence "Edition des notes obtenues pour un contrôle".

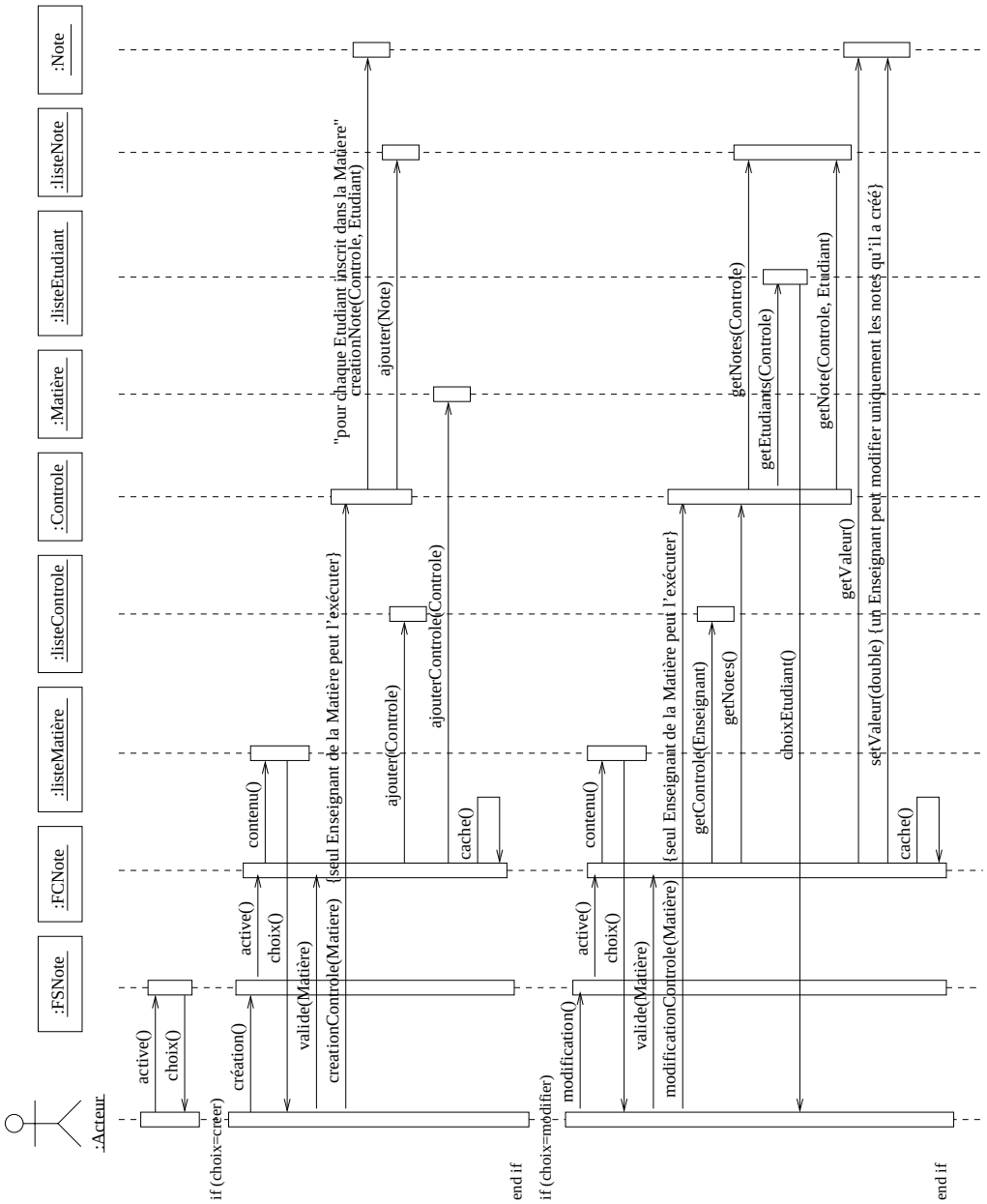


FIG. A.15 – Le diagramme de séquence "Saisie de Note".

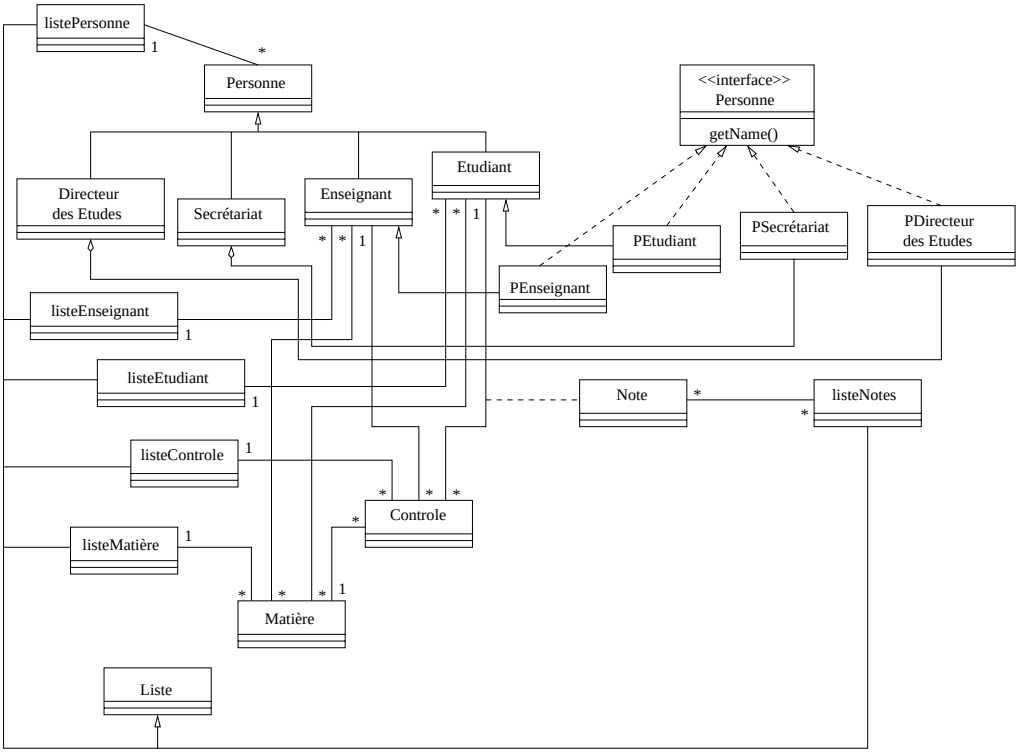


FIG. A.16 – Le diagramme de classes de l'application "Gestion des Notes".

Annexe B

Le méta-modèle UML

Le méta-modèle d'UML est décomposé en trois paquetages de haut niveau : Behavioral Elements, Foundation et Model Management [Gro97, Gro00a].

Nous présentons dans cette annexe seulement les diagrammes de paquetages qui sont utilisés par la suite de notre travail.

Le paquetage *Behavioral Elements* définit les concepts nécessaires pour modéliser le comportement du système. Il se compose de quatre autres paquetages (Figure B.1) :

1. *Common Behavior* - spécifie des concepts essentiels pour le comportement des éléments. Il définit une structure de base nécessaire pour les éléments dynamiques (Figure B.2).
2. *Collaborations* - spécifie un contexte comportemental nécessaire à la réalisation d'une tâche particulière (Figure B.3 et Figure B.4).
3. *Use Cases* - spécifie le comportement en utilisant les acteurs et les use cases (Figure B.5).
4. *State Machines* - définit le comportement du système en utilisant un système d'état-transition. Il spécifie comment contrôler un flux de données entre les éléments d'un modèle pour réaliser un cas d'utilisation.

Le paquetage *Foundation* définit les concepts nécessaires pour une modélisation structurelle et les mécanismes pour étendre le langage. Ces éléments de modélisation permettent de modéliser la structure d'un système. Ce paquetage se compose aussi de quatre paquetages (Figure B.1) :

1. *Data Types* - définit les structures de base des données du langage.
2. *Core* - spécifie les concepts de base nécessaires pour réaliser un méta-modèle (Figure B.6, Figure B.7 et Figure B.8).
3. *Auxiliary Elements* - définit des constructions additionnelles pour étendre le paquetage Core pour qu'il puisse supporter des concepts avancés.
4. *Extension Mechanisms* - spécifie comment des éléments du modèle peuvent être étendus avec de nouvelles sémantiques.

Le paquetage *Model Management* (Figure B.9) spécifie une organisation des éléments dans les modèles, les paquetages et les systèmes. Il définit comment les éléments de modélisation sont organisés.

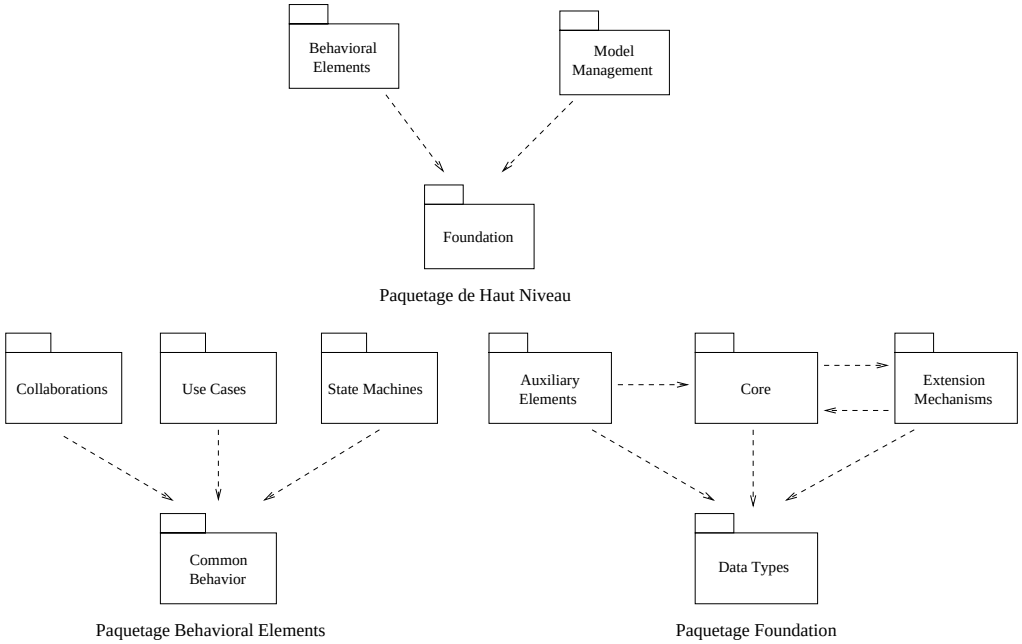


FIG. B.1 – Les paquetages de haut niveau d'UML.

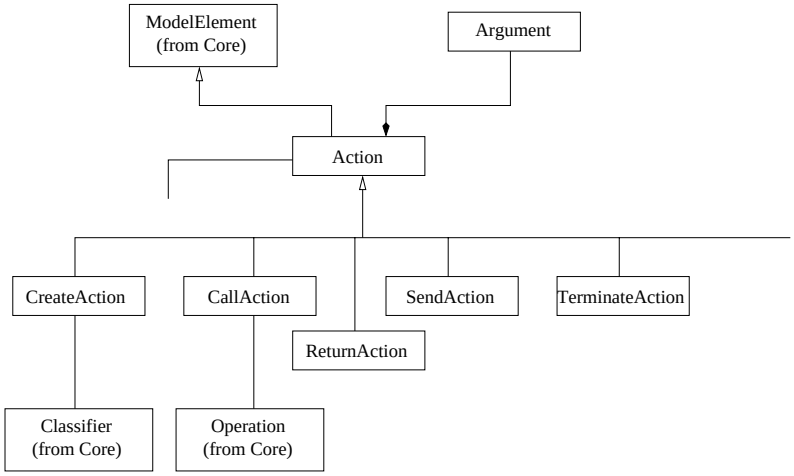


FIG. B.2 – L'extrait du paquetage Common Behavior - Actions (de Behavioral Elements).

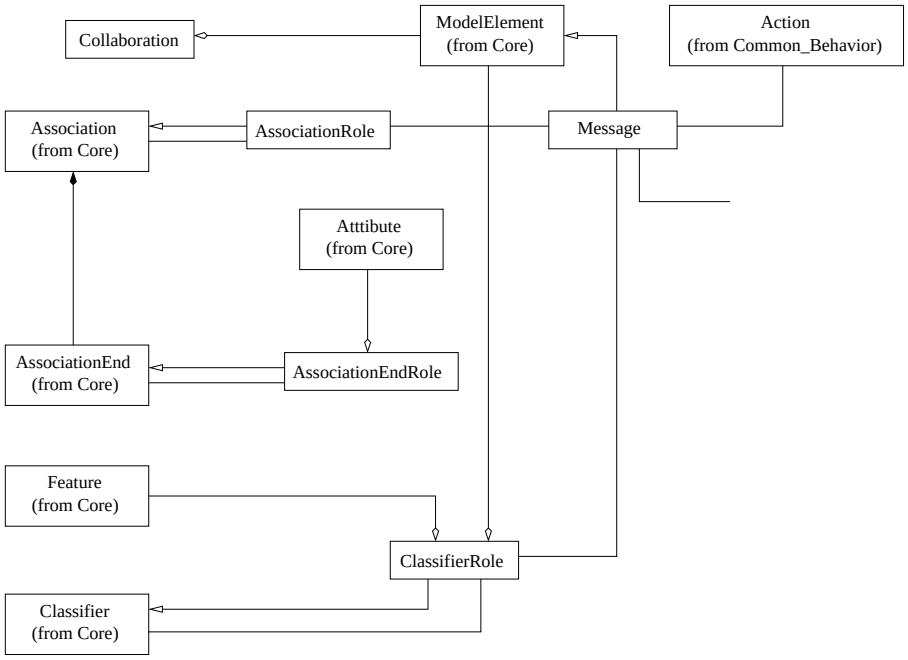


FIG. B.3 – L'extrait du paquetage Collaborations - Roles (de Behavioral Elements).

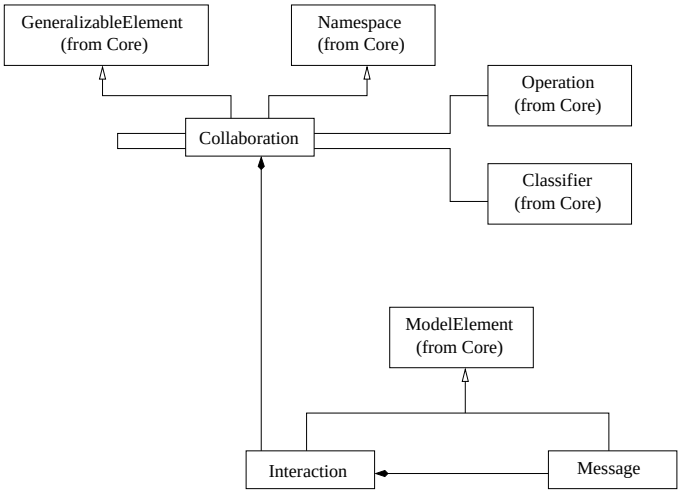


FIG. B.4 – L'extrait du paquetage Collaborations - Interactions (de Behavioral Elements).

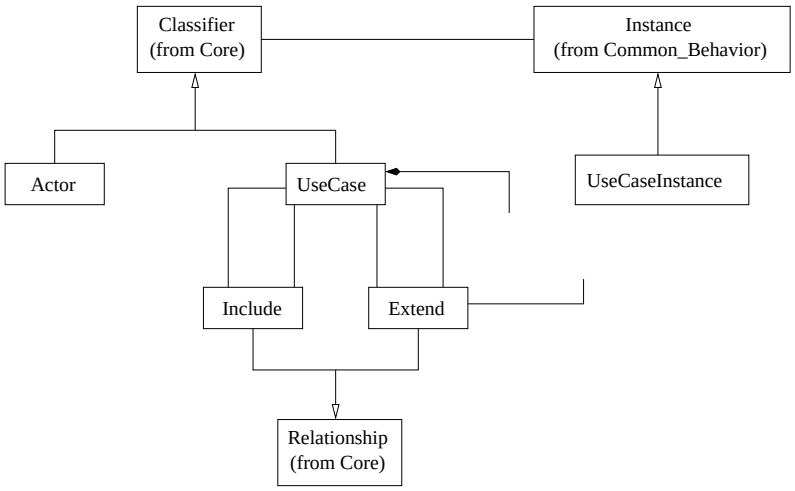


FIG. B.5 – L'extrait du paquetage Use_Cases (de Behavioral Elements).

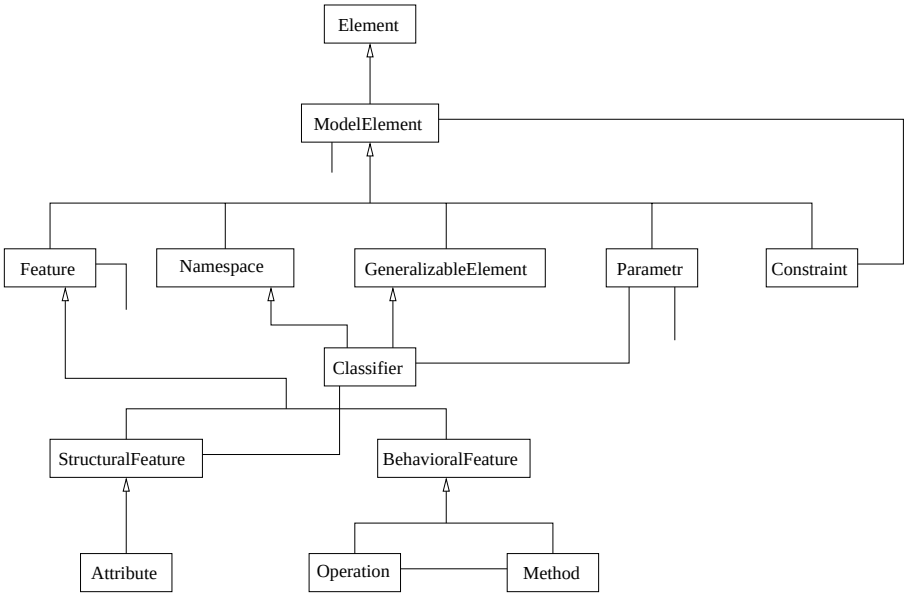


FIG. B.6 – L'extrait du paquetage Backbone (de Core).

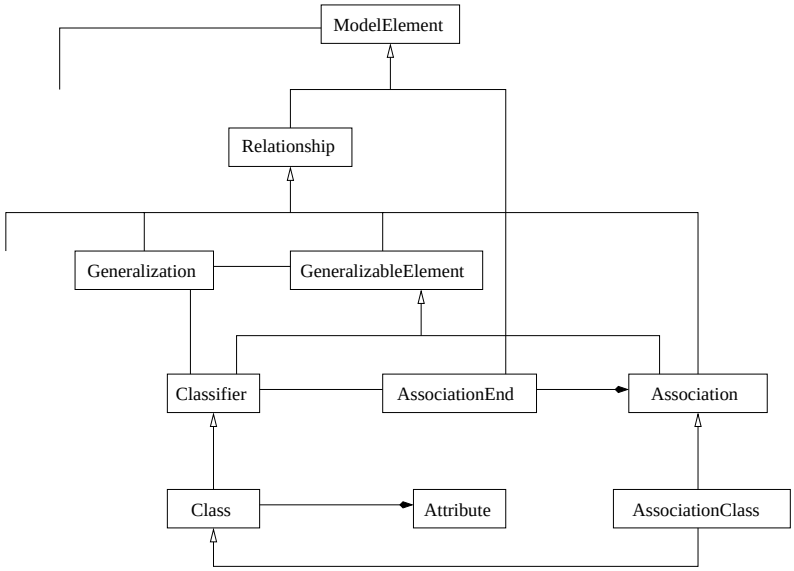


FIG. B.7 – L'extrait du paquetage Relationships (de Core).

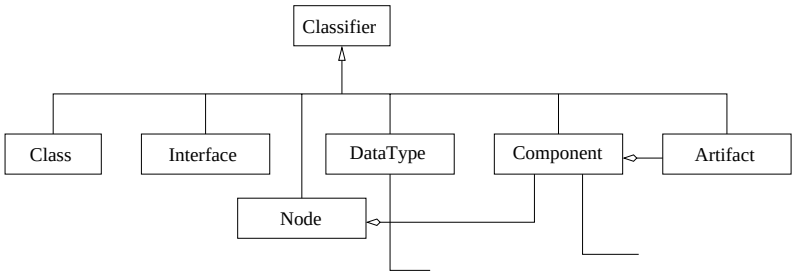


FIG. B.8 – L'extrait du paquetage Classifiers (de Core).

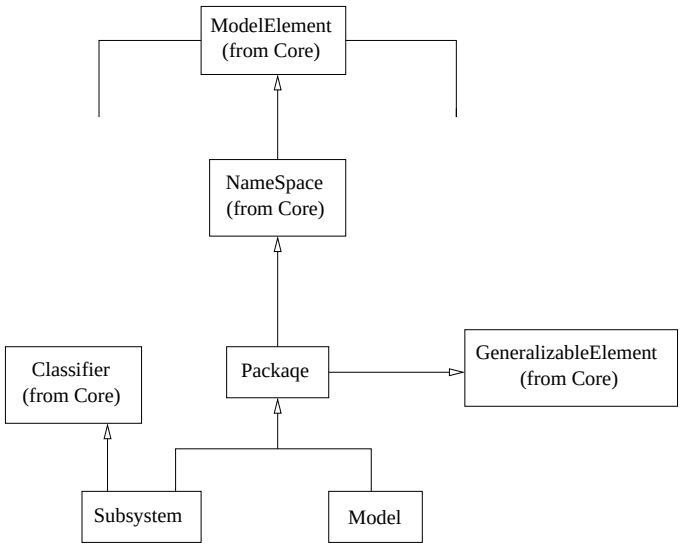


FIG. B.9 – L'extrait du paquetage ModelManagement.

Annexe C

La méthode de traduction des contraintes d'OCL vers RCL 2000 étendu

Nous présentons la description détaillée de la méthode de traduction des contraintes décrite dans le Chapitre 4. La traduction est proposée pour trois types de contraintes qui peuvent être définies au niveau du concepteur de l'application : les contraintes sur une permission, les contraintes de pré-requis et les contraintes de cardinalité.

C.1 La traduction des contraintes sur une permission

On propose la méthode de traduction des contraintes sur une permissions du langage OCL au langage RCL 2000 étendu.

En OCL
Les contraintes sur une permission peuvent être exprimées en OCL sous forme de pré-condition :

context Classe : : Methode
pre : objet | objetCondition.

Une condition *objetCondition* est exprimée par une expression booléenne. Elle peut contenir une ou des expressions séparées par un opérateur logique :

$$objetCondition ::= expression\ (operateurLogique\ expression)*$$

$$operateurLogique ::= ' and' \mid ' or' \mid ' implies'$$

Une expression *expression* contient une ou des sous-expressions séparées par un opérateur relationnel :

$$expression ::= sousExpression\ (operateurRelationnel\ sousExpression)*$$

$$operateurRelationnel ::= '>' \mid '\geq' \mid '<' \mid '\leq' \mid '=' \mid '<>'$$

Une sous-expression *sousExpression* est composée d'une ou de plusieurs expressions élémentaires séparées par un opérateur arithmétique :

$$sousExpression ::= elementExpressionOCL\ (operateurArithmètique\ elementExpressionOCL)*$$

$$operateurAritmètique ::= ' +' \mid ' -' \mid ' *' \mid ' /'$$

Une *elementExpressionOCL* est composée d'un ou de plusieurs éléments et éventuellement avec une opération définie en OCL :

$$elementExpressionOCL ::= element\ (('.' \mid '\rightarrow')\ element \mid operationOCL)*$$

avec :

$$element ::= objet \mid methode \mid classe$$

et *operationOCL* est une des opérations définies dans le langage OCL, par exemple *oclIsTypeOf()* ou *size()*.

Le point et la flèche sont utilisés pour la navigation. Une valeur de propriété d'un objet est spécifiée par un point suivi du nom de cette propriété ou d'une opération d'OCL comme, par exemple *oclIsTypeOf()*. Une propriété d'une collection est accessible par une flèche suivie du nom de cette propriété ou d'une opération d'OCL, par exemple *size()* ou *select()*.

$$operationOCL\ pour\ les\ objets = \{OclAny,\ oclType,\ oclIsKindOf,\ oclIsTypeOf,\ oclAsType,\ ...\}$$

$$operationOCL\ pour\ les\ ensembles =$$

$$= \{count,\ sum,\ size,\ iterate,\ forAll,\ select,\ isEmpty,\ notEmpty,\ includes,\ exists,\ ...\}$$

En RCL 2000 étendu

En utilisant le langage RCL 2000 on peut écrire :

$$p\,(m,\ o) = OE\,(P)$$

Pour cette permission il faut déterminer un ensemble d'objets sur lequel un utilisateur peut exécuter la méthode. L'ensemble d'objets est décrit en général comme suit :

$$O' = \{o \mid class\,(o) = C\ \wedge\ expressionB\,(o) = vrai\}$$

$$expressionB\ \text{- une expression booléenne qui représente le prédicat}\ \pi$$

Un ensemble O' contient les objets qui sont de la classe C et pour lesquels la condition $expressionB\,(o)$ est vraie.

Cette condition peut se composer de plusieurs sous-conditions et cela de manière récursive. En se basant sur la syntaxe du langage RCL 2000 [Ahn99], on présente cette expression booléenne *expressionB* sous la forme suivante :

$$expressionB ::= expression\ (operateurLogique\ expression)*$$

où

$$operateurLogique ::= '\wedge' \mid '\vee' \mid '\Rightarrow'$$

Une expression *expression* contient une ou deux sous-expressions séparées par un opérateur relationnel :

$$expression ::= sousExpression\ (operateurRelationnel\ sousExpression)?$$

où

$$operateurRelationnel ::= '>' \mid '\geq' \mid '<' \mid '\leq' \mid '=' \mid '<>'$$

Une sous-expression *sousExpression* est composée d'une ou de plusieurs expressions élémentaires séparées par un opérateur :

$$sousExpression ::= elementExpression\ (operateur\ elementExpression)*$$

$$operateur ::= '\cap' \mid '\cup' \mid '*' \mid '/'$$

Une expression élémentaire *elementExpression* est composée d'une fonction du langage RCL 2000 étendu ou d'une expression qui calcule la dimension d'un ensemble :

$$elementExpression ::= fonctionRCL2000(argument\,(,\ argument)?)\ |$$

$$('|'\ fonctionRCL2000(argument\,(,\ argument)?)'\ |')\ |\ size\ |\ ('|'\ ensemble'\ |')\ |'\in'$$

$$size ::= \emptyset \mid 1 \mid 2 \mid ... \mid N$$

$$ensemble ::= U \mid R \mid F \mid P \mid M \mid OP \mid O \mid C \mid S$$

$fonctionRCL2000$ représente une des fonctions définies dans le langage RCL 2000 étendu (avec les quantificateurs $OE()$ et $AO()$) et

$$argument ::= ensemble \mid fonctionRCL2000(argument\,(,\ argument)?) \mid argument$$

D'OCL à RCL 2000 étendu

Les concepts de classe, de méthode et d'objet en OCL sont identiques aux concepts de classe, de méthode et d'objet dans le langage RCL 2000 étendu qui ont été déjà présentés comme suit :

$$traductionClasse\,(Classe\ en\ OCL) \rightarrow c \in C\ en\ RCL2000$$

$$traductionMethode\,(Methode\ en\ OCL) \rightarrow m \in M\ en\ RCL2000$$

$$traductionObjet\,(objet\ en\ OCL) \rightarrow o \in O\ en\ RCL2000$$

où *traductionClasse*, *traductionMethode* et *traductionObjet* représentent les méthodes de traduction d'une classe, d'une méthode et d'un objet de OCL en RCL 2000 étendu.

Une condition sur l'objet *objetCondition* qui détermine une contrainte en OCL peut être traduite en prédicat exprimé par une expression booléenne en RCL 2000 étendu :

$$traductionCondition\,(objetCondition\ en\ OCL) \rightarrow expressionB()\ en\ RCL2000$$

La condition *objetCondition* possède les éléments suivants qui peuvent être traduits en RCL 2000 étendu :

1. *expression* en OCL vers *expression* en RCL 2000 étendu

   $$traduction\,(expression\ en\ OCL) \rightarrow expression\ en\ RCL2000$$

   les opérateurs logiques de l'ensemble *operateurLogique* d'OCL traduits : un opérateur *and* vers "$\wedge$", un opérateur *or* vers "$\vee$" et un opérateur *implies* vers "$\Rightarrow$" de RCL 2000 étendu,
2. *sousExpression* en OCL vers *sousExpression* en RCL 2000 étendu

   $$traduction\,(sousExpression\ en\ OCL) \rightarrow sousExpression\ en\ RCL2000$$

   $$traduction\,(operateurRelationnel\ en\ OCL) \rightarrow operateurRelationnel\ en\ RCL2000$$
3. *elementExpressionOCL* en OCL vers *elementExpression* en RCL 2000 étendu

   $$traduction\,(elementExpressionOCL\ en\ OCL) \rightarrow elementExpression\ en\ RCL2000$$

   $$traduction\,(operateurAritmètique\ en\ OCL) \rightarrow operateur\ en\ RCL2000$$

   L'expression *elementExpressionOCL* en OCL peut prendre l'une des formes suivantes :

(a) $elementExpressionOCL ::= element\ ('.'element)*\ ('\rightarrow'\ element)*$ peut être traduite en utilisant des fonctions du langage RCL 2000 étendu vers

$$elementExpression ::= fonctionRCL2000(argument\ (,\ argument)?)$$

où $fonctionRCL2000$ représente une des fonctions définies dans le langage RCL 2000 étendu (avec les quantificateurs $OE()$ et $AO()$) et

$$argument ::= ensemble \mid fonctionRCL2000(argument\ (,\ argument)?) \mid argument$$

(b) $elementExpressionOCL ::= element\ ('.'element)*\ ('.'operationOCL)*$ peut être traduite en utilisant des fonctions du langage RCL 2000 comme dans le cas (a) et de plus les opérations d'OCL testent dans ce cas le type d'un élément vers

$$fonctionRCL2000(argument\ (,\ argument)?) \in fonctionRCL2000(argument\ (,\ argument)?)$$

(c) $elementExpressionOCL ::= element\ ('.'element)*\ ('\rightarrow'\ operationOCL)*$ peut être traduite en utilisant des fonctions du langage RCL 2000 comme dans le cas (a). Les opérations d'OCL sont traduites comme suit :

i. *count*, *sum*, *size* vers

$$('|'\ fonctionRCL2000(argument\ (,\ argument)?)'\ |') \mid size \mid ('|'\ ensemble'\ |')$$

où $size ::= \emptyset \mid 1 \mid 2 \mid ... \mid N$ et $ensemble ::= U \mid R \mid F \mid P \mid M \mid OP \mid O \mid C \mid S$,

ii. *iterate*, *forAll* et *select* vers

$$fonctionRCL2000(argument\ (,\ argument)?)$$

iii. *isEmpty* vers

$$('|'\ fonctionRCL2000(argument\ (,\ argument)?)'\ |') \mid ('|'\ ensemble'\ |') = \emptyset$$

et *notEmpty* vers

$$('|'\ fonctionRCL2000(argument\ (,\ argument)?)'\ |') \mid ('|'\ ensemble'\ |') \neq \emptyset$$

iv. *includes*, *exists* vers

$$fonctionRCL2000(argument\ (,\ argument)?) \mid ensemble\ \in$$

$$fonctionRCL2000(argument\ (,\ argument)?) \mid ensemble$$

(d) $elementExpression ::= element\ ('\rightarrow'\ element)*\ ('\rightarrow'\ operationOCL)*$ peut être traduite en utilisant des fonctions du langage RCL 2000 comme dans le cas (c).

La traduction des éléments qui sont contenus dans *element*

$$element ::= objet \mid methode \mid classe$$

donc des concepts *objet*, *méthode*, *classe* de OCL vers RCL 2000 étendu se résume à une réécriture sans traitement particulier. On utilise les fonctions de RCL 2000 étendu :

$$traduction(objet, cst) \rightarrow object(p),\ p \in P$$

$$traduction(methode, cst) \rightarrow methods(f, o),\ f \in F\ et\ o \in O$$

$$traduction(classe, cst) \rightarrow class(o),\ o \in O$$

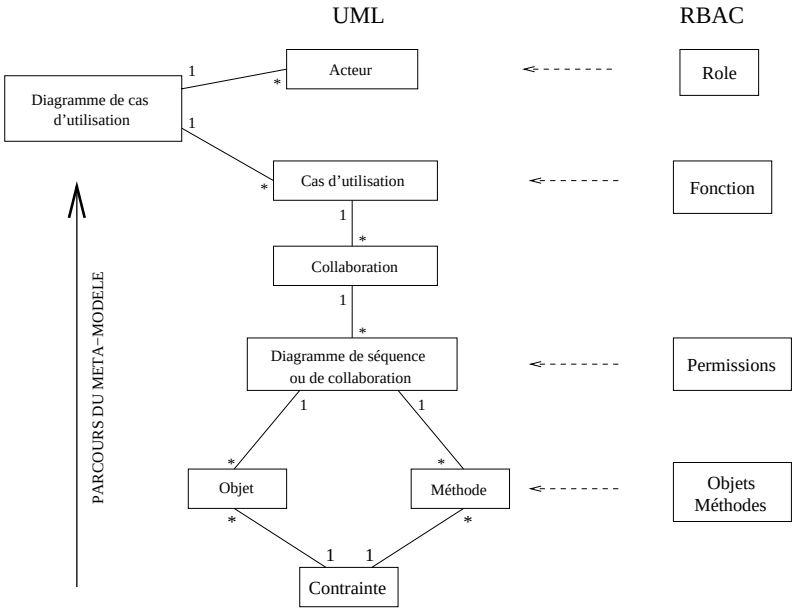


FIG. C.1 – Le parcours du méta-modèle pour identifier les contraintes.

où *cst* est l'identification d'une contrainte appliquée à *objet*, *méthode* ou *classe*.

Ces fonctions de RCL 2000 étendu possèdent des argument et il faut déterminer les arguments de la contrainte correspondante en OCL.

La fonction *object()* possède un argument : une permission du modèle. Une permission est identifiée par un couple : une méthode et un ensemble d'objets sur lesquels la méthode peut être exécutée. La méthode et l'ensemble des objets existent dans la définition de la contrainte (Figure C.1).

La fonction *methods()* possède deux arguments : une fonction du modèle et un objet du modèle. Dans le langage UML, un parcours du méta-modèle nous permet d'identifier les contraintes. A partir d'une contrainte on peut retrouver le cas d'utilisation (fonction) dans lequel la méthode contrainte est utilisée (Figure C.1). L'objet est quant à lui identifie dans la définition de la contrainte en OCL.

La fonction *class()* possède un argument : un objet du modèle. Un objet est identifie comme un objet en OCL.

La présentation d'une contrainte d'OCL en RCL 2000 étendu :

$$object\,(OE\,(P)) = object\,(p\,(m, o)) = O'$$

Nous avons montré comment traduire les éléments contenu dans une contrainte sur une permission du langage OCL vers le langage RCL 2000 étendu.

C.2 La traduction des contraintes de pré-requis

On présente la méthode de traduction des contraintes de pré-requis du langage OCL au langage RCL 2000 étendu.

En OCL

Les contraintes de pré-requis sont exprimées en OCL sous forme d'invariants (présentée dans le Chapitre 4). La forme de l'expression *Condition* peut être décomposée en deux parties :

$$Condition = Condition1\ implies\ Condition2$$

et la forme générale de ce type de contraintes est comme suit :

context Classe inv :
Condition1 implies Condition2

La classe *Classe* est un élément qui peut être de type :

$$Classe ::= acteur \mid use\ case \mid methode \mid objet$$

Les conditions *Condition1* et *Condition2* peuvent avoir la forme suivante :

$$Condition1,\ Condition2 ::= (instances\,(Classe\,(element1*)) \rightarrow includes\,(Set\,(element2*)))$$

où

$$element1,\ element2 ::= use\ case \mid methode \mid objet$$

Ces conditions, *Condition1* et *Condition2*, vérifient si un ensemble d'instances d'une *Classe* contient les éléments précisés de type *cas d'utilisation*, *méthode* ou *objet* (les éléments qui appartiennent à un ensemble $element1,\ element2 ::= use\ case \mid methode \mid objet$). Les conditions utilisent la fonction standard d'OCL - *includes()*.

Les éléments qui doivent être identifiés et traduit sont les suivants : acteur, cas d'utilisation, méthode et objet.

En RCL 2000 étendu

Les éléments qui sont importants dans ce type de contraintes sont les rôles, les fonctions et les permissions. Ils peuvent être obtenus par les fonctions du langage RCL 2000 étendu : ***roles*** et ***fonctions***. Une *permission* a été définie dans la section C.1.

D'OCL à RCL 2000 étendu

La traduction d'un acteur *Acteur*, d'un cas d'utilisation *UseCase*, d'une méthode *Methode* et d'un objet *objet* du langage OCL en RCL 2000 étendu est la suivante :

$$traductionActeur\,(Acteur\ en\ OCL) \rightarrow r \in R\ en\ RCL2000$$

$$traductionUseCase(UseCase\ en\ OCL) \rightarrow f \in F\ en\ RCL2000$$

$$traductionMethode\,(Methode\ en\ OCL) \rightarrow m \in M\ en\ RCL2000$$

$$traductionObjet\,(objet\ en\ OCL) \rightarrow o \in O\ en\ RCL2000$$

où *traductionActeur*, *traductionUseCase*, *traductionMethode* et *traductionObjet* représentent les méthodes de traduction d'un acteur, d'un cas d'utilisation, d'une méthode et d'un objet d'OCL en RCL 2000 étendu.

Il faut lier les concepts d'OCL avec ceux de RCL 2000 étendu :

- **pour les fonctions pré-requis :**
  OCL :
  context Acteur inv :
  $self.UseCase \rightarrow includes\,(uc1)\ implies\ self.UseCase \rightarrow includes\,(uc2)$
  RCL 2000 étendu :
  $$f1 \in functions\,(OE\,(R)) \ \Rightarrow\ f2 \in functions\,(OE\,(R))$$
- **pour les permissions pré-requis :**
  OCL :
  context UseCase inv :
  $self.permission \rightarrow includes\,(p1)\ implies\ self.permission \rightarrow includes\,(p2)$
  ou
  context UseCase inv :

$$self.permission \rightarrow includes\,(p\,(methode1,\ objet1))\ implies$$

$$self.permission \rightarrow includes\,(p\,(methode2,\ objet2))$$

RCL 2000 étendu :

$$p1 \in permissions\,(OE\,(F))\ \Rightarrow\ p2 \in permissions\,(OE\,(F))$$

ou

$$p\,(methode1,\ objet1) \in permissions\,(OE\,(F)) \Rightarrow p\,(methode2,\ objet2) \in permissions\,(OE\,(F))$$

C.3 La traduction des contraintes de cardinalité

On présente la traduction des contraintes de cardinalité du langage OCL au RCL 2000 étendu.

En OCL

Les contraintes de cardinalité sont exprimées en OCL sous forme d'invariants (présentée dans le Chapitre 4). La forme de l'expression *Condition* peut être présentée comme suit :

$$Condition ::= elementCondition\ implies\ numCondition$$

et la forme générale de ce type de contraintes est comme suit :

context Classe inv :
elementCondition implies numCondition

Les conditions *elementCondition* et *numCondition* représentent des expressions booléennes. La présentation de ces expressions en OCL est la même que la condition *objetCondition* déterminée dans la section C.1. On peut écrire :

$$elementCondition = condition\ et\ numCondition = condition$$

$$condition ::= expression\ (operateurLogique\ expression)*$$

Les différences apparaissent pour une *elementExpression* :

$$elementExpression ::= (element\ (('.' \mid '\rightarrow')\ element \mid operationOCL)*) \mid nombre$$

$$element ::= acteur \mid use\ case \mid classe \mid methode \mid objet$$

$$nombre \in \{1, 2, ..., N\}$$

Les éléments qui doivent être identifiés et traduits sont les suivants : classe, elementCondition et numCondition.

En RCL 2000 étendu

Les éléments qui sont importants dans ce type de contrainte sont les rôles, les fonctions, les permissions, les méthodes et les objets. Ils peuvent être obtenus par les fonctions correspondantes de RCL 2000 étendu.

D'OCL à RCL 2000 étendu

La traduction d'une classe *Classe* du langage OCL en RCL 2000 étendu :

$$traductionClasse\,(Classe\ en\ OCL) \rightarrow c \in C\ en\ RCL2000$$

La traduction des conditions *elementCondition* et *numCondition* peut être réalisée de la même manière que la traduction d'une condition sur l'objet *objetCondition* présentée dans la section C.1. Donc, on traduit ces conditions vers un prédicat exprimé par une expression booléenne en RCL 2000 étendu :

$$traductionCondition\,(elementCondition\ en\ OCL) \rightarrow expressionB()\ en\ RCL2000$$

$$traductionCondition\,(numCondition\ en\ OCL) \rightarrow expressionB()\ en\ RCL2000$$

Donc les conditions *elementCondition* et *numCondition* exprimées en OCL sont traduites vers *expressionB* en RCL 2000 de la même manière que dans la section C.1. Des différences apparaissent au moment de la traduction des concepts les plus simples :

$$element ::= acteur \mid use\ case \mid classe \mid methode \mid objet$$

La traduction des éléments qui sont contenus dans *element* est réalisée en utilisant les fonctions de RCL 2000 étendu :

$$traduction(Acteur, cst) \rightarrow roles(f),\ f \in F$$

$$traduction(UseCase, cst) \rightarrow fonctions(r),\ r \in R\ ou\ fonctions(p),\ p \in P$$

$$traduction(objet, cst) \rightarrow object(p),\ p \in P$$

$$traduction(methode, cst) \rightarrow methods(f, o),\ f \in F\ et\ o \in O$$

$$traduction(classe, cst) \rightarrow class(o),\ o \in O$$

où *cst* est l'identification d'une contrainte.

La fonction *roles()* possède un argument : une fonction du modèle. A partir d'une contrainte on peut retrouver la fonction dans laquelle la méthode contrainte est utilisée. Une fonction est identifiée comme un cas d'utilisation dans lequel existe une méthode avec cette contrainte (Figure C.1).

La fonction *fonctions()* possède un argument : un rôle du modèle ou une permission du modèle. Un rôle est identifié comme un acteur qui est lié avec un cas d'utilisation dans lequel existe une méthode avec la contrainte. Une permission est identifiée un couple : une méthode et un ensemble d'objets sur lesquels la méthode peut être exécutée. La méthode et l'ensemble des objets existent dans la définition de la contrainte (Figure C.1).

Les fonctions *object()*, *methods()* et *class()* ont été présentées dans la section C.1.

Annexe D

Le langage XML

XML, abréviation de ***eXtensible Markup Language***, est un langage de description et d'échange de documents structurés. Il permet de décrire la structure logique de documents textuels, à l'aide d'un système de balises permettant de marquer les éléments qui composent la structure et les relations entre ces éléments [Mic99, Gro00b].

D.1 Le document réalisé en XML

XML distingue deux types de documents : les documents bien formés et les documents valides [Mic99].

Un *document est bien formé* quant il obéit aux règles syntaxiques du langage XML. Il sera traité avec succès par un processeur adéquat et par un parseur XML. Document bien formé est donc synonyme de document correct.

Un *document valide* est bien formé et il obéit en outre à une structure type définie explicitement dans une DTD (Document Type Definition). Il est nécessaire que tout document valide peut-être distribué dans un système d'information (par exemple Web) sans DTD.

D.1.1 La structure d'un document

Le syntaxe de document XML contient :
- un *prologue* - facultative mais conseillée ; contient un certain nombre de déclarations,
- un *arbre d'éléments* - forme le contenu proprement dit du document,
- des *commentaires* et des *instructions de traitement* - facultatives et peuvent apparaître aussi bien dans le prologue que dans l'arbre d'éléments.

D.1.1.1 Le prologue

Le prologue peut contenir une déclaration XML, des instruction de traitement et une déclaration de type de document.

La **déclaration XML** possède la forme suivante :

< ?xml version="1.0" encoding='ISO-8859' ?>

elle indique au processeur qui va traiter le document :
- la version de langage XML utilisée dans le document,
- le codage de caractères utilisés dans le document, si la valeur par défaut ne convient pas.

La **déclaration de type de document** indique dans le cas où le document se conforme à une structure type particulière, quel est ce type :

< !DOCTYPE livre SYSTEM "livre.dtd" [déclarations] >

Cet exemple de la déclaration de type indique que le document qui la contient est de type "livre" et qu'il se conforme à la structure type définie dans la ressource "livre.dtd".

D.1.1.2 L'arbre d'éléments

Chaque élément d'un document XML se compose d'une *balise d'ouverture*, d'un contenu d'élément et d'une *balise de clôture* :

<nom>contenu de l'élément</nom>

Le nom d'un élément figurant dans les balises d'ouverture et de clôture est formé de caractères alphanumériques, du tiret-souligné, du signe moins ou du point. Aucun nom ne peut pas commencer par la chaîne "xml" qu'elle soit en minuscules, majuscules ou mélange des deux.

La partie essentielle d'un document est formée d'une hiérarchie d'éléments qui dénote la structure sémantique de son contenu. Par exemple un livre sera décrit sous forme d'une hiérarchie incluant un titre, une liste d'acteurs, une date de publication et une suite de chapitres. Tous ces éléments forment ensemble une hiérarchie unique - **un arbre**. Un document XML doit contenir un arbre d'éléments et les conséquences de cette restrictions sont comme suite :

- tout élément fils est complètement inclus dans son père, il ne peut pas y avoir recouvertement d'éléments,
- il existe dans un document un et un seul élément qui contient tous les autres - un élément racine.

D.1.1.3 Les commentaires

Des commentaires peuvent être inclus dans un document. Ils sont réalisés par les marques de début et de fin de commentaire :

< !– des commentaires –>

D.1.2 Les éléments et les attributs

Une balise d'ouverture d'un élément se compose de l'élément, puis éventuellement d'un ou plusieurs attributs utilisés pour décrire certaines propriétés de l'élément. Chaque attribut est une paire : *nom = 'valeur'*. Le nom d'un attribut obéit aux mêmes règles que les noms d'élément.

Un élément peut contenir d'autres éléments, des données, des références à des entités, des sections littérales et des instructions de traitement.

Il est aussi possible de définir ses propres entités et d'y faire référence dans le contenu des éléments. Leur déclaration se fait dans la déclaration de type de document, alors avant le première élément du document. Chaque déclaration d'entité associe un nom à une valeur de remplacement.

D.2 Les types de structures du document

Document Type Declaration (DTD) - ensemble de déclarations définissant des noms d'éléments et leurs attributs ainsi que leur règle d'association. Les DTDs sont une possibilité pour définir la structure de documents XML.

Un document valide respecte la structure qui s'applique à tout document XML. Pour être valide un document doit inclure dans son prologue une déclaration de type et cette déclaration de type doit contenir des déclarations pour tous les éléments, attributs, entités qui seront utilisés dans le document.

Une DTD est un ensemble de toutes les déclarations contenues dans une déclaration de type de document **DOCTYPE**.

Il est aussi possible de définir certaines déclarations dans le document et de faire en autre appel à une DTD contenue dans une entité externe. Les déclarations locales au document forment la partie interne de la DTD et celles contenues dans l'entité externe forment sa partie externe. Pour cela réaliser on crée deux fichiers : *exemple.dtd* et *exemple.xml*.

D.2.1 Le contenu d'une DTD

Un document DTD peut contenir des éléments suivants :
- déclarations d'éléments,
- déclarations de listes d'attributs,
- déclarations d'entités,
- commentaires.

La **déclaration d'un élément** définit un type d'élément et associe un nom de type à un modèle de contenu :

<!ELEMENT nom modèle>

Tous les éléments instances de ce type qui apparaîtront dans un document devront avoir un contenu conforme au modèle défini dans la déclaration. Le modèle contenu peut autoriser la création d'éléments qui contiendront :
- un ou plusieurs éléments fils spécifiés,
- des données représentées par un flot de caractères,
- un mélange de données et d'éléments fils spécifiés.

Les éléments fils peuvent avoir un ordre imposé et forment une séquence. Ils peuvent aussi être en ordre libre. La séquence est spécifiée par la liste des noms de types d'éléments séparés par des virgules :

<!ELEMENT chapitre(titre, intro, section) >.

Un nom d'élément apparaissant dans le modèle d'un élément père peut être suffixé par un indicateur d'occurrence, représenté par l'un des caractères :
- **?** - élément peut apparaître zéro ou une fois dans le contenu d'une instance, il est optionnel et non répétable,
- ***** - élément peut apparaître zéro, une ou plusieurs fois dans le contenu d'une instance, il est optionnel et répétable,
- + - élément peut apparaître une ou plusieurs fois dans le contenu d'une instance, il est obligatoire et répétable.

L'absence de l'un de ces suffixes signifie que l'élément doit apparaître une et une seule fois et il est obligatoire et non répétable.

La déclaration suivante :

<!ELEMENT chapitre(titre, intro, section+) >.

permet dans un document instance de créer la structure :

<chapitre>
<titre>...</titre>
<intro>...</intro>
<section>...</section>
<section>...</section>
</chapitre>.

La présence de données dans le contenu d'un élément est indiquée par le mot-clef ***#PCDATA*** (Parsed Character Data) :

<!ELEMENT titre (#PCDATA)>

Selon cette déclaration, un élément 'titre' ne peut pas contenir que des données.
Il est fréquent de définir des modèles mixtes autorisant de mêler des données et des éléments :

<!ELEMENT p(#PCDATA | em |exposant)>.*

Un élément peut être spécifié comme obligatoirement vide en utilisant le mot-clef **EMPTY**.

La déclaration de la liste des attributs
La déclaration d'attribut dans un DTD permet de spécifier quels sont les attributs qui pourront être associés à une instance d'élément et d'indiquer éventuellement quelle est une valeur par défaut d'un attribut :

<!ATTLIST nom-élément nom-attribut type-attribut déclaration-de-défaut>.

Le *type d'attribut* peut être :
- **CDATA** - la valeur de l'attribut est une chaîne de caractères,
- **ID** ou **IDREF** - on peut définir des renvois à l'intérieur des documents,
- **ENTITY** ou **ENTITIES** - un attribut prend comme une valeur le nom d'une entité externe non XML,
- **NMTOKEN** ou **NMTOKENS** - abréviation de *Name token*, il permet à l'attribut de prendre comme valeur le nom d'un token,
- une liste de choix possibles dans un ensemble de "tokens", c'est-à-dire de noms symboliques obéissant à la syntaxe des noms XML.

La *déclaration de défaut* peut prendre quatre formes suivantes :
- simplement la valeur par défaut de l'attribut,
- #**REQUIRED** - un attribut est obligatoirement dans chaque instance d'élément du type déclaré,
- #**IMPLIED** - la présence de l'attribut dans une instance de l'élément est facultative,
- #**FIXED** 'valeur' - un attribut prend la même valeur dans toute instance de l'élément déclaré.

D.3 Les avantages du langage XML

L'écriture de document DTD et XML est complexe, au même titre que l'écriture de programmes. On peut présenter quelques règles générales de la construction :
- modularité du document - il faut définir dans des entités séparées les parties réutilisables, modifiables indépendamment les unes des autres,
- regrouper les déclarations d'entités paramètres en tête de la DTD,
- simplicité des modèles de contenu - il est nécessaire de définir des modèles de contenu simples et lisibles,
- bonne utilisation des commentaires - il faut utiliser des commentaires de façon concise mais précise pour : documenter une DTD en indiquant son historique, son auteur, son usage, etc., pour matérialiser le début et la fin de la DTD et pour aider la lecture de la DTD,
- adéquation aux besoins d'une classe d'utilisateurs bien - on doit éviter d'écrire des DTD très générales, supposées permettre la création des documents les plus divers ; c'est mieux d'avoir plusieurs DTD spécifiées pour des types de documents plus spécifiés.

Annexe E

Les documents XML

Il existe deux types de fichiers qui forment le document résultat, le fichier de type DTD et le fichier de type XML.

Un fichier DTD a été appelé *RBAC.dtd* et il est comme suit :

<?xml version = '1.0' ?>
<!ELEMENT RBAC(role+, fonction+, permission+, methode+, objet+, operation+, classe+)>
<!ELEMENT role(nom, refRole*, refFonction+)>
<!ATTLIST role id ID #REQUIRED>
<!ELEMENT nom(#PCDATA)>
<!ELEMENT refRole EMPTY>
<!ATTLIST refRole idRef IDREF #REQUIRED>
<!ATTLIST refRole type(inheritance) #REQUIRED>
<!ELEMENT refFonction EMPTY>
<!ATTLIST refFonction idRef IDREF #REQUIRED>
<!ELEMENT fonction(nom, refRole+, refFonction*, refPermission+)>
<!ATTLIST fonction id ID #REQUIRED>
<!ELEMENT refFonction EMPTY>
<!ATTLIST refFonction idRef IDREF #REQUIRED>
<!ATTLIST refFonction type(include | extend | inheritance) #REQUIRED>
<!ELEMENT refPermission EMPTY>
<!ATTLIST refPermission idRef IDREF #REQUIRED>
<!ELEMENT permission(refMethode+, refObjet+, refFonction+)>
<!ATTLIST permission id ID #REQUIRED>
<!ELEMENT refMethode EMPTY>
<!ATTLIST refMethode idRef IDREF #REQUIRED>
<!ELEMENT refObjet EMPTY>
<!ATTLIST refObjet idRef IDREF #REQUIRED>
<!ELEMENT methode(nom, refPermission+, refOperation+, attribute*)>
<!ATTLIST methode id ID #REQUIRED>
<!ELEMENT refOperation EMPTY>
<!ATTLIST refOperation idRef IDREF #REQUIRED>
<!ELEMENT attribute(#PCDATA)>
<!ELEMENT objet(nom, refPermission*, refClass+, baseClass)>
<!ATTLIST objet id ID #REQUIRED>
<!ELEMENT refClass EMPTY>
<!ATTLIST refClass idRef IDREF #REQUIRED>
<!ELEMENT baseClass(#PCDATA)>
<!ELEMENT operation(nom, refMethode*)>

<!ATTLIST operation id ID #REQUIRED>
<!ELEMENT class(nom, refObjet*)>
<!ATTLIST class id ID #REQUIRED>

E.1 La description du fichier RBAC.dtd

On a défini au début un élément principal qui s'appelle ***RBAC*** et qui peut contient plusieurs éléments : des rôles, des fonctions, des permissions, des méthodes, des objets, des opérations et des classes.

Un élément ***rôle*** contient un nom, une référence à un autre rôle (refRole) et à une fonction (refFonction) et il possède un attribut appelé ***id*** qui détermine une indication pour les autres éléments du modèle. Un élément ***nom*** contient un nom d'un élément. Un élément ***refRole*** décrit une relation entre les deux rôles et il donne un type de cette relation en utilisant un attribut ***type*** - ici cette relation est une relation d'héritage (inheritance). Une relation entre un rôle et une fonction est représentée par un élément ***refFonction***.

Un élément ***fonction*** représente une fonction dans un modèle RBAC étendu, il possède quatre éléments : ***nom*** qui donne un nom de cette fonction, ***refRole*** qui représente une relation entre un rôle et une fonction, refFonction et refPermission. Cet élément fonction possède aussi un attribut : ***id*** qui détermine une indication pour les autres éléments du modèle. Un élément ***refFonction*** détermine une relation entre les deux fonctions, il contient deux attributs : ***idRef*** et ***type*** qui donne un type de cette relation - une relation d'utilisation (include), d'extension (extend) ou d'héritage (inheritance). Un élément ***refPermission*** représente une relation entre une fonction et une permission.

Un élément ***permission*** définit une permission du modèle RBAC, il contient trois éléments : refMethode, refObjet et refFonction. De la même façon, ***refMethode*** représente une relation entre une permission et une méthode. Un élément ***refObjet*** est une relation entre une permission et un objet. Et un élément ***refFonction*** représente une relation entre une fonction et une permission.

Un élément ***methode*** se compose de quatre éléments : ***nom*** qui donne le nom de cette méthode, ***refPermission*** qui représente une relation entre une méthode et une permission, ***refOperation*** qui détermine une relation entre une méthode et une opération et ***attritute*** qui définit un ou plusieurs attributs de cette méthode.

Un élément ***objet*** représente un objet du modèle RBAC. Il se compose de quatre éléments : ***nom*** - un nom d'un objet, ***refPermission*** - une relation entre un objet et une permission, ***refClass*** - une relation entre un objet et sa classe et ***baseClass*** qui donne une classe de base pour cet objet.

Un élément ***operation*** représente une opération du modèle RBAC et il possède deux éléments : ***nom*** - un nom de cette opération et ***refMethode*** qui représente les relations entre cette opération et ses méthodes.

Un dernier élément ***class*** représente une classe du modèle et il possède deux éléments : ***nom*** - un nom de cette classe et ***refObjet*** qui représente les relations entre cette classe et ses objets.

E.2 La description du fichier RBAC_NomApplication.xml

Le deuxième fichier, RBAC.xml contient les éléments concrets qui existent dans l'application du système d'information. Nous présentons la structure du fichier RBAC_GestionDesNotes.xml de l'application "Gestion des Notes".

La partie qui concerne la description d'un rôle peut être la suivante :

<role id = S.10007>
<nom>Enseignant</nom>
<refFonction idRef = S.10011/>
<refFonction idRef = S.10012/>

</role>

La première ligne donne un numéro de cet élément dans un système (*id*), la deuxième - son nom (dans cet exemple : Enseignant) et ensuite les relations entre ce rôle et des fonctions, représentées par des indicateurs *refFonction* ; il existe pour chaque indicateur un numéro de fonction avec laquelle ce rôle est en relation.

La partie qui touche la description d'une fonction :

<fonction id S.10012>
<nom>visualiser</nom>
<refRole idRef = S.10007/>
<refRole idRef = S.10009/>
<refRole idRef = S.10010/>
<refFonction idRef = S.10015/>
<refFonction type = include/>
</fonction>

La première ligne donne aussi un numéro de cet élément dans un système, la deuxième - son nom (dans cet exemple : visualiser). On a les indicateurs *refRole*, qui représentent les relations entre cette fonction et autres rôles en donnant les numéros de ces rôles. Un indicateur *refFonction* représente une relation entre cette fonction et autre fonction donnée par son numéro. Pour cette relation on a aussi son type (dans cet exemple : une relation d'utilisation).

La partie de la description d'une permission :

<permission id = P.1>
<refMethode idRef = G.17/>
<refObjet idRef = G.14/>
<refFonction idRef = S.10011/>
</permission>

On a le numéro de cette permission et une méthode indiquée par un indicateur *refMethode* qui peut être exécutée sur un objet indiqué par un indicateur *refObjet*. La dernière ligne présente une relation entre cette permission et une fonction en utilisant un élément *refFonction* avec un numéro de cette fonction.

La partie qui concerne la description d'une méthode est la suivante :

<methode id = G.17>
<nom>contenu()</nom>
<refPermission idRef = P.1/>
<refOperation idRef = O.1/>
<attribute></attribute>
</methode>

La première ligne donne le numéro de cette méthode, la deuxième - son nom (dans cet exemple : contenu()). Après il y a une relation entre cette méthode et sa permission, indiquée par un numéro de cette permission. La ligne suivante donne une opération de cette méthode et ensuite la liste des attributs de la méthode (dans cet exemple la méthode n'a pas d'attributs).

La partie de la description d'un objet peut être la suivante :

<objet id = G.16>
<nom>unEnseignant</nom>
<refPermission idRef = P.7/>
<refPermission idRef = P.15/>
<refClass idRef = S.10007/>

<baseClass>Enseignant</baseClass>
</objet>

On a au début le numéro de cet objet, après son nom (dans cet exemple : unEnseignant), des relations entre cet objet et les permissions, indiquées par ses indicateurs *refPermission*. Ensuite il y a une classe de base de cet objet. Cette classe est représentée par son numéro en utilisant un indicateur *refClass* et par son nom (dans cet exemple une classe de base est : Enseignant).

La partie qui touche la description d'une opération :

<operation id = O.1>
<nom>contenu()</nom>
<refMethode idRef = G.17/>
</operation>

La première ligne donne le numéro de cette opération, la deuxième son nom et après il y a un ou plusieurs méthodes de cette opération, indiquées par des indicateurs *refMethode*.

La partie de la description d'une classe peut être la suivante :

<class id = S.10001>
<nom>Enseignant</nom>
<refObjet idRef = G.14/>
</class>

On a au début le numéro de cette classe, après son nom et un ou plusieurs objets de cette classe indiqués par des indicateurs *refObjet*.

Bibliographie

[AA92] R. Ahad and All. Supporting access control in an object-oriented database language. In *3rd International ConferenceExtending Database Technology*, 1992.

[Ahn99] G.-J. Ahn. *The RCL 2000 Language for Specifying Role-Based Authorization Contraintes*. PhD thesis, George Mason University, 1999.

[Arg] http ://argouml.tigris.org/.

[AS99] G.-J. Ahn and R.S. Sandhu. The rsl99 language for role-based separation of duty constraints. *Proceedings of 4th ACM Workshop on Role-Based Access Control*, 1999.

[Bib77] K. J. Biba. Integrity considerations for secure computer systems. In *The MITRE Corp.*, 1977.

[BP75] D. E. Bell and L. J. La Padula. Secure computer systems : unified exposition and multics interpretation. In *The MITRE Corp.*, 1975.

[BRJ98] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modelling Language User Guide*. Addison Wesley, 1998.

[BS97] V. Bhamidipati and R. Sandhu. The arbac97 model for role-based administration of roles. In *2nd ACM Workshop on Role-Based Access*, 1997.

[CFMS94] S. Castaro, M. G. Fugini, G. Martella, and P. Samarati. *Database Security*. Addison-Wesley, 1994.

[Clu] http ://www.clusif.asso.fr.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. In *ACM Comminications*, 1970.

[Cod76] E. F. Codd. The entity-relationship model towards a unified view of data. *ACM*, 1976.

[CS95] F. Chen and R.S. Sandhu. Contraintes for role-based access control. In *Proceedings for 1st ACM Workshop on Role-Based Access Control*, 1995.

[CS97] J.-L. Chappelet and J.-J. Snella. *Un langage pour l'organisation, L'approche Ossad*. Presses Polytechniques et Universitaires Romandes, 1997.

[ES99] P. Epstein and R. S. Sandhu. Towards a uml based approach to role engineering. In *Proceedings of 4th ACM Worksho on Role-Based Access Control, Virginia*, 1999.

[FBK99] D. F. Ferraiolo, J. F. Berkley, and D. R. Kuhn. A role-based access control model and reference implementation within a corporate internet. In *ACM Transactions on Information ans System Security*, 1999.

[FCK95] D. Ferraiolo, J. Cugini, and R. Kuhn. Role-based access control (rbac) : Features and motivations. In *Proceedings of 11th Annual Computer Security Application Conference*, 1995.

[FH97] E. B. Fernandez and J. C. Hawkins. Determining role rights from use cases. In *Proceedings of 2nd ACM Workshop on Role-Based Access Control*, 1997.

[FK92] D. Ferraiolo and R. Kuhn. Role-based access controls. In *Proceedings of 15th NIST-NCSC National Computer Security Conference*, 1992.

[FLLP91] B. France-Lanord, V. Lahalle, and G. Petit. *Organisation et direction de l'entreprise.* Masson, 1991.

[FM84] M. G. Fugini and G. Martella. Acten : a conceptual model for security system desing. In *Computers and Security.* Elsevier, 1984.

[Gab98] J. Gabay. *Merise vers OMT et UML.* InterEditions, 1998.

[GH00a] S. Ghernaouti-Hélie. *Sécurité Internet, Stratégies et technologies.* Dunod, 2000.

[GH00b] G. Goncalves and F. Hemery. From use case in uml to the roles management in information system. In *Proceedings of SNPD, Reims - AICS*, 2000.

[GH00c] G. Goncalves and F. Hemery. Une plate-forme uml-xml pour la gestion des rôles d'un système d'information. In *Proceedings of Inforsid, Lyon*, 2000.

[GH01] S. Ghernaouti-Hélie. *Stratégie et ingénierie de la sécurité des réseaux.* Dunod, 2001.

[Gro97] Object Management Group. *The Unified Modeling Language*, release 1.1., reference manual edition, 1997.

[Gro00a] Object Management Group. *OMG Unified Modeling Language Specification*, 2000.

[Gro00b] Object Management Group. *OMG XML Metadate Interchange Specification*, 2000.

[Gui95] L. Guiri. A new model for role-based access control. In *Proceedings of 11th Annual Computer Security Application Conference*, 1995.

[HRU76] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Comm. ACM*, 1976.

[Jav] http ://java.sun.com.

[JLS76] A. K. Jones, R. J. Lipton, and L. Snyder. A linear time algorithm for deciding security. In *Proc. of 17th Annual Symposium on Foundations of Computer Science*, 1976.

[JSpSB98] S. Jajodia, V. S. Subrahmanian, p. Samarati, and E. Bertino. An unified framework for enforcing multiple access control policies. In *Proceedings ACM SIGMOD Conference of Management of Data*, 1998.

[Ken97] S. Kent. Constraint diagrams : Visualizing invariants in object-oriented models. In *Proceedings of OOPSLA97, ACM Press.*, 1997.

[KMPRS98] N. Kettani, D. Mignet, P. Paré, and C. Rosenthal-Sabroux. *De Merise à UML.* Eyrolles, 1998.

[Kru95] Ph. Kruchten. The 4 + 1 view model of architecture. *IEEE Software*, 1995.

[Lam71] B. W. Lampson. Protection. In *Proceedings of 5th Symposium on Information Sciences and Systems*, 1971.

[Lam91] J. M. Lamere. *Sécurité des systèmes d'information, Comment organiser la sécurité et la qualité des systèmes d'information dans l'entreprise.* Dunod, 1991.

[Mai02] E. Maiwald. *Sécurité des réseaux.* CampusPress, 2002.

[Mic99] A. Michard. *XML, langage et applications.* Eyrolles, 1999.

[Mél72] J. Mélése. *L'Analyse Modulaire des Systèmes.* AMS, Editions Hommes et Techniques, 1972.

[Mof98] J. Moffett. Control principles and role hierarchies. In *3rd ACM Workshop on Role-Based Access Control*, 1998.

[Moi77] J. L. Le Moigne. *La théorie du système général : théorie de la modélisation.* PUF, Paris, 1977.

[Mul98] P. A. Muller. *Modélisation objet avec UML.* Eyrolles, 1998.

[Mun00] Q. Munawer. *Administrative Models for Role-Based Access Control.* PhD thesis, George Masson University, 2000.

[PG99] A. Poniszewska and G. Goncalves. Graphical tool for the security management in information systems based on the uml notation. *Journal of Applied Computer Science*, 1999.

[PGH01] A. Poniszewska, G. Goncalves, and F. Hemery. Implementation of the rbac model on the uml-java platform. In *Proceedings of the SMC Conference, Poland*, 2001.

[Pon01] A. Poniszewska. Modelling of information system security based on rbac model. In *Proceedings of 5th National Conference - Diagnistic of Industrial Process, Poland*, 2001.

[Rat] http ://www.rational.com/uml/.

[Rei99] R. Reix. *Systèmes d'information et management des organisations*. Vuibert, 1999.

[Res00] Response to omg action semantics for the uml. Technical report, 2000.

[RG] M. Richters and M. Gogolla. A semantics for ocl pre- and postconditions.

[San90] R. S. Sandhu. Separation of duties in computerized information systems. In *Proc. of the IFIP WG 11.3 Workshop on Database Security, Halifax*, 1990.

[San93] R. S. Sandhu. Lattice-based access control models. *IEEE Computer*, 1993.

[San94] R. S. Sandhu. *Handbook of Information Security Management*, chapter Relational Database Access Control. Auerbach Publishers, 1994.

[San96] R. S. Sandhu. Role hierarchies and constraints for lattice-based access control. In *Proc. Fourth European Symposium of Research in Computer Security, Rome, Italy*, 1996.

[San98] R. S. Sandhu. Role activation hierarchies. In *Proceedings of the 3rd ACM Workshop on Role-Based Access Control, Fairfax*, 1998.

[SaQM99] R. S. Sandhu and V. Bhamidipati ang Q. Munawer. The arbac97 model for role-based administration fo roles. In *ACM Transactions on Information and Systems Security*, 1999.

[SB97] R. S. Sandhu and V. Bhamidipati. The ura97 model for role-based user-role assignment. In *Proceeding of IFIP WG 11.3 Workshop on Database Security, Lake Tahoe*, 1997.

[SCFY94] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control : A multi-dimensional view. In *Proc. of 10th Annual Computer Security Application Conf., Ontario, Florida*, 1994.

[SCFY96] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 1996.

[SFK00] R. S. Sandhu, D. Ferraiolo, and R. Kuhn. The nist model for role-based access control : Towards a unified standard. In *ACM RBAC*, 2000.

[SJ93] R. S. Sandhu and S. Jajodia. *Handbook of Information Security Management*, chapter Data and Database Security and Controls. Auerbach Publishers, 1993.

[SM98a] R. S. Sandhu and Q. Munawer. How to do discretionary access control using roles. In *Proceeding of 3nd ACM Workshop on Role-Based Access Control, Fairfax*, 1998.

[SM98b] R. S. Sandhu and Q. Munawer. The rra97 model for role-based administration of role hierarchies. *Proceedings of 14th Annual Computer Security Applications Conference, Arizona*, 1998.

[SM99] R. S. Sandhu and Q. Munawer. The arbac99 model for administration of roles. *Proceedings of INFOSECU99 International Conference on Information Security, China*, 1999.

[SS] S. Sendall and A. Strohrneier. Enhancing ocl for specifying pre- and postconditions.

[SS94] R. S. Sandhu and P. Samarati. Access control : Principles and practice. *IEEE Communication*, 1994.

[SS97] R. S. Sandhu and P. Samarati. Authentication, access control and intrusioin detection. In *The Computer Science and Engineering hand-book*, 1997.

[TNH75] H. Tardieu, D. Nanci, and H. Heckennoth. Méthode, modèle et oitil pour la conception de la base de données d'un système d'information. Technical report, Centre d'études techniques et l'équipement, Aix-en-Provence, 1975.

[TOB98] D. Thomsen, D. O'Brien, and J. Bogle. Role based access control framework for network enterprises. In *14 th Annual Computer Security Application Conference*, 1998.

[TRC86] H. Tardieu, A. Rochfield, and R. Colletti. *La Méthode Merise, Principles et outils.* Editions d'organisation, 1986.

[TS93] R. K. Thomas and R. S. Sandhu. Discretionary access control in object-oriented databases : Issues and research directions. *Proc. of the 16th NIST-NCSC National Computer security Conference, Baltimore*, 1993.

[TS94] R. Thomas and R. S. Sandhu. Conceptual foundations for a model of task-based authorizations. In *Proceedings of the 7th IEEE Computer Security Foundations Workshop*, 1994.

[WK99a] J. B. Warmer and A. G. Kleppe. *The object constraint language. Precise modelling with UML.* Addison - Wesley, 1999.

[WK99b] J. Wermer and A. Kleppe. Ocl : the constraint language of the uml. *Journal of Object-Oriented Programming*, 1999.

[XMI] http ://www.omg.org/technology/documents/spec-catalog.htm.

[XML] http ://www.w3.org/xml/.

Résumé

De nos jours avec la mondialisation des marchés et l'accélération des échanges d'information qui en découlent, le système d'information d'une entreprise devient de plus en plus stratégique pour la pérennité de celle-ci. Il importe donc de sécuriser efficacement celui-ci. Une facette de la sécurité consiste à assurer l'accès logique aux différents composants (i.e. données, services) du système d'information.

Nous avons choisi d'aborder le problème du contrôle d'accès d'un système d'information en proposant un modèle de rôles dès la conception de celui-ci et tout au long de son évolution (i.e. ajouts de nouvelles applications). Nos objectifs étaient d'une part de faciliter le travail de l'administrateur de sécurité et d'autre part d'avoir une meilleure cohérence entre les contraintes globales de sécurité de l'entreprise et les différents composants de son système d'information.

Pour ce faire, nous avons utilisé une conception orientée objet décrite dans le langage UML. Grâce à ces diagrammes de haut niveau, UML permet de spécifier clairement les besoins du système d'information et facilite le dialogue entre les différents acteurs (i.e. concepteur, utilisateur, administrateur ...). C'est pourquoi nous l'avons privilégié et de plus il est devenu depuis le démarrage de cette thèse un standard de facto dans la communauté objet.

A partir donc d'une spécification UML fourni par un concepteur, nous avons montré comment il était possible de générer automatiquement les rôles associés à un composant du système d'information ceci en utilisant le méta-modèle d'UML. Pour cela nous avons rapproché certains concepts d'UML avec ceux de notre modèle RBAC étendu.

Nous avons ensuite intégré la notion de contrainte d'accès dans les diagrammes UML à l'aide du langage OCL pour spécifier des contraintes applicatives et nous avons utilisé au niveau de l'administrateur de sécurité le langage RCL 2000 pour spécifier des contraintes plus globales. Nous avons proposé un algorithme pour unifier l'ensemble des contraintes vers le langage RCL 2000 et un autre algorithme pour vérifier une cohérence "minimale" du modèle RBAC lors de l'intégration d'un nouveau composant dans un système d'information existant.

Mots clefs : système d'information, sécurité, contrôle d'accès, rôle, conception de rôles, UML, RBAC